João Paulo de Oliveira Libório

Licenciado em Engenharia Informática

Privacy-Enhanced Dependable and Searchable Storage in a Cloud-of-Clouds

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Henrique João Lopes Domingos, Full Professor, NOVA University of Lisbon

Júri

Presidente: Nuno Manuel Ribeiro Preguiça
Arguentes: Alysson Neves Bessani
Henrique João Lopes Domingos

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2016

Privacy-Enhanced Dependable and Searchable Storage in a Cloud-of-Clouds

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to thank my thesis advisor, Prof. Henrique João Domingos, for the help, support and advise during the preparation and elaboration of this thesis and the teaching staff at Faculdade de Ciências e Tecnologia who teached me a lot over the course. I would also like to thank my family for the support and encouragement over the years, my friends for being there in the good and bad times and my colleagues for making these last years memorable.

ABSTRACT

In this dissertation we will propose a solution for a trustable and privacy-enhanced storage architecture based on a multi-cloud approach. The solution provides the necessary support for multi modal on-line searching operation on data that is always maintained encrypted on used cloud-services. We implemented a system prototype, conducting an experimental evaluation. Our results show that the proposal offers security and privacy guarantees, and provides efficient information retrieval capabilities without sacrificing precision and recall properties on the supported search operations.

There is a constant increase in the demand of cloud services, particularly cloud-based storage services. These services are currently used by different applications as outsourced storage services, with some interesting advantages. Most personal and mobile applications also offer the user the choice to use the cloud to store their data, transparently and sometimes without entire user awareness and privacy-conditions, to overcome local storage limitations. Companies might also find that it is cheaper to outsource databases and key-value stores, instead of relying on storage solutions in private data-centers. This raises the concern about data privacy guarantees and data leakage danger. A cloud system administrator can easily access unprotected data and she/he could also forge, modify or delete data, violating privacy, integrity, availability and authenticity conditions.

A possible solution to solve those problems would be to encrypt and add authenticity and integrity proofs in all data, before being sent to the cloud, and decrypting and verifying authenticity or integrity on data downloads. However this solution can be used only for backup purposes or when big data is not involved, and might not be very practical for on-line searching requirements over large amounts of cloud stored data that must be searched, accessed and retrieved in a dynamic way. Those solutions also impose high-latency and high amount of cloud inbound/outbound traffic, increasing the operational costs. Moreover, in the case of mobile or embedded devices, the power, computation and communication constraints cannot be ignored, since indexing, encrypting/decrypting and signing/verifying all data will be computationally expensive.

To overcome the previous drawbacks, in this dissertation we propose a solution for a trustable and privacy-enhanced storage architecture based on a multi-cloud approach, providing privacy-enhanced, dependable and searchable support. Our solution provides the

necessary support for dependable cloud storage and multi modal on-line searching operations over always-encrypted data in a cloud-of-clouds. We implemented a system prototype, conducting an experimental evaluation of the proposed solution, involving the use of conventional storage clouds, as well as, a high-speed in-memory cloud-storage backend. Our results show that the proposal offers the required dependability properties and privacy guarantees, providing efficient information retrieval capabilities without sacrificing precision and recall properties in the supported indexing and search operations.

Keywords: Data storage clouds, dependable cloud storage, multi cloud architectures, multi modal searchable encryption, RAM storage.

Resumo

Nos últimos anos tem-se verificado um aumento notável da utilização de plataformas e serviços na nuvem, incluindo os serviços de armazenamento e gestão de dados. Estes serviços são hoje amplamente utilizados e integrados em muitas aplicações, como componentes outsourcing, tendo em vista as suas vantagens. As aplicações pessoais e de utilização móvel, que manipulam cada vez mais dados e informação multimédia usam também aqueles serviços, para armazenamento e acesso remoto de informação, de forma transparente para o utilizador e como forma de ultrapassar limitações de armazenamento dos dispositivos móveis. Em muitos casos os utilizadores desses dispositivos não tem mesmo consciência desse facto. Por outro lado, muitas empresas recorrem cada vez mais a serviços de gestão e armazenamento de dados na nuvem, como forma de reduzir custos operacionais bem como ultrapassar limitações de investimento em infra-estruturas próprias e centros de dados privados.

A anterior realidade, no entanto, colide com um conjunto de preocupações também crescente, em relação às reais garantias de privacidade, integridade, disponibilidade e fiabilidade de dados críticos, quando estes estão armazenados fora do completo controlo do utilizador. De facto, um administrador de sistemas trabalhando para um provedor de serviços na nuvem pode, por exemplo, violar aquelas garantias, aumentando o perigo de revelação de dados privados e pondo em causa as condições de confiabilidade que se esperaria estarem preservadas. Não são completamente controláveis pelo utilizador as reais condições de segurança e controlo de acesso aos dados depositados na nuvem ou que garantias existem face a ataques externos, por exploração de vulnerabilidades do software ou das plataformas de hardware/-software que suportam os serviços disponibilizados. São várias as situações de incidentes e ataques que têm sido reportadas ao longo do tempo.

Uma solução possível para mitigar ou resolver alguns dos problemas anteriores poderia passar por garantir que todos os dados fossem cifrados, sendo acopladas provas de autenticidade e integridade, antes de serem enviados para repositórios na nuvem. Os dados também poderiam ser arquivados ou salvaguardados com base na sua replicação ou fragmentação em múltiplas nuvens, operadas por diferentes provedores de serviços independentes. Neste último caso o acesso aos dados pressupõe a sua localização, leitura e descarregamento de modo a serem acedidos pelos utilizadores correctos e decifrados nos seus próprios dispositivos, considerando-se estes confiáveis. Esse tipo de soluções só é interessante para salvaguarda

ou acesso de pequenos volumes de dados, não sendo adequados quando é necessário pesquisar e recuperar "on line"grandes quantidades de dados cifrados, nomeadamente no caso da pesquisa, acesso e recuperação de dados multimodais. Por outro lado, as anteriores soluções impõem naturalmente elevada latência de acesso, com aumento muito significativo de tráfego trocado entre o cliente e as nuvens, com agravamento de custos operacionais e redução da efectividade e desempenho das aplicações. No caso de dispositivos móveis ou dispositivos embebidos, os custos de computação, comunicação e de consumo energético devidos a operações de treino, indexação e localização dos dados, descarregamento, cifra e decifra, aquelas soluções são inutilizáveis.

Para ultrapassar as anteriores limitações, a presente dissertação propõe uma solução de armazenamento confiável e com garantias de privacidade acrescida, com base num sistema de armazenamento de dados pesquisáveis, numa arquitectura de nuvem de nuvens. A solução avançada permite suportar operações de escrita, leitura, indexação e pesquisa de dados num contexto multimodal, sendo os dados mantidos sempre cifrados nas nuvens que os albergam e permitindo que as pesquisas sejam feitas na nuvem usando apenas dados cifrados.

O sistema proposto foi implementado e validado experimentalmente, tendo sido testado para suportar dados em contexto de utilização multimodal (texto e imagem), incluindo-se o suporte de pesquisas por similaridade. A avaliação inclui o estudo do impacto da solução quando a mesma usa diferentes soluções de armazenamento confiável com base em soluções de provedores Internet, e quando se utiliza um repositório na nuvem para armazenamento em memória (do tipo in memory storage cloud). Os resultados obtidos mostram a validade da proposta e que é possível oferecer garantias de confiabilidade e privacidade acrescidas com repositórios indexáveis e pesquisáveis na nuvem, sendo possível suportar técnicas de indexação e recuperação eficiente de dados multimodais mantidos cifrados. A avaliação experimental revela que se conseguem suporta pesquisas eficientes, sem perda de precisão e sem que se ponha em causa a correcção da recuperação desses dados.

Palavras-chave: Clouds de armazenamento, armazenamento seguro, arquitecturas multi cloud, cifras pesquisáveis multi modais, armazenamento em memória ...

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

INTRODUCTION

In this chapter we introduce the context and motivation as well as the problem statement for this thesis, followed by the goals and contributions. In the end we present the structure for the following chapters.

1.1 Context and Motivation

In recent years the use of cloud services has been increasing exponentially. Among the biggest increases in usage is the upload and sharing of visual data, like photos and videos. This increase is easily seen with the growth of Instagram in the last years [1]. Outsourcing that data to the cloud is an attractive approach since it has several advantages: it's cheaper than hosting the data on a company server; has some guarantees of persistence since cloud providers backup their data regularly and has a dynamic allocation of resources so that a company only uses, and pays for, what it needs. Although managing visual data is harder and requires more resources than textual data, searching for a specific image might be easier done by the content and characteristics of the image rather than user defined expressions.

Unfortunately outsourcing data to the cloud, although attractive at first sight given the recognized advantages of cloud storage solutions, has its own set of problems. Data owners and users cannot control all guarantees of privacy over the outsourced data and in many circumstances lose the expected privacy or confidentiality, as reported in many real incidents of cloud-based data-leakage and confidentiality breaks.

Privacy is a strong concern for applications managing sensitive data. As data will be stored on the provider infrastructure using their native storage solutions managed out of the entire control of users, it will be more vulnerable. A malicious system administrator or external attacker that somehow gain access to the system, by exploiting vulnerabilities in access-control services or in the software and operating systems provided by each specific

cloud-vendor [2] could have complete access to the data. As a consequence, those opponents could leak or modify the state of data-repositories, by data tampering, adding or deleting actions. Indeed, different cloud services could have vulnerabilities (sometimes "zero-day vulnerabilities"as extensively reported), or could not be trusted in terms of the integrity of software stacks supporting the provided services for their end-users. The client has no way to audit the cloud infrastructure and confirm the data integrity conditions or to attest the correctness of data-access operations. Cloud providers also have downtimes, also reported in many real situations, which make data unavailable and breaking the expected availability conditions [3].

Although it may seem like all these concerns are not realistic, events have shown otherwise. Google admitted that people shouldn't expect privacy on GMail [4], a Google engineer spied on chats [5] and the US government had access to servers from many of the major Internet companies like Microsoft, Google, Apple, Facebook [6]. External threads are also happening, as it was the case of iCloud photo leakage [7] and more recently Instagram "million dollar bug". Although there was no leak of data in the Instagram case as it was reported to Facebook's Bug Hunting program [8], the bug would give an external attacker access to private keys used to authenticate session cookies and connections allowing them to impersonate any valid system user (having access to all private data from that user), including Instagram staff or the server itself [9]. Moreover, the decision to use outsourced data-storage solutions, should lead users to reflect carefully on the expected dependability conditions. To illustrate the concern we can take (as one of many representative examples) the real conditions expressed in typical statements for the provision of such services, by a reference Internet cloud-service provider:

Excerpts from the liability statement and customer agreement conditions by Amazon (Amazon Web Services™ Customer Agreement Documentation) for the provision of Amazon AWS resources and services, namely on Security and Data Privacy conditions [1]:

> *Security. We strive to keep Your Content secure, but cannot guarantee that we will be successful at doing so, given the nature of the Internet. Accordingly, without limitation to Section 4.3 above and Section 11.5 below, you acknowledge that you bear sole responsibility for adequate security, protection and backup of Your Content. We strongly encourage you, where available and appropriate, to use encryption technology to protect Your Content from unauthorized access and to routinely archive Your Content. We will have no liability to you for any unauthorized access or use, corruption, deletion, destruction or loss of any of Your Content.*

[1] http://portal.aws.amazon.com/gp/aws/developer/terms-and-conditions.html

Data Privacy. (....) we will implement reasonable measures designed to help you secure your content against accidental or unlawful loss, access or disclosure. (....) You may specify the AWS regions in which Your Content will be stored. You consent to the storage of Your Content in, and transfer of Your Content into, the AWS regions you select. We will not access or use Your Content except as necessary to maintain or provide the Service Offerings (....) but you accept the necessary conditons to comply with the law or a binding order of a governmental body. We will not (a) disclose Your Content to any government or third party or (b) subject to Section 3.3, move Your Content from the AWS regions selected by you; except in each case as necessary to comply with the law or a binding order of a governmental body (....) in accordance with the Privacy Policy you accept.

1.2 Problem Statement

With all the problems that outsourcing data to the cloud raises, many companies and users are wary of their use. This doesn't mean however that cloud providers shouldn't be used, but instead that solutions should be found as a way to fully utilize cloud services to their potential. Although some solutions can seem easy and solve some of the problems, they might also raise other, and harder, problems that need to be solving. One perfect example is the privacy of the data outsourced.

One possible approach to the lack of privacy guarantees on data outsourced to the cloud is to encrypt it before outsourcing it, assuming that the client is trusted. This however raises the problem that operations must be done in the client. Although in some cases the overhead of retrieving, decrypting, processing, re-encrypting and uploading the data might be small if the client knows exactly which data to retrieve and that data is small, operations like searching or operations on large quantities of data become impractical. Even for small quantities of data this approach is limiting for lightweight client devices like smartphones or tablets. The solution would be to outsource the computations to the cloud and operate on encrypted data. Although solutions for executing operations on encrypted data exist by using fully homomorphic encryption they are computationally too expensive [10]. Other schemes like CBIR presented by Ferreira et al [11] are more promising as they allow searches on encrypted multi-modal data with a much smaller overhead.

On top of privacy concerns there is also integrity and availability concerns. Data stored on the cloud is vulnerable to attacks on the cloud itself. It will become unavailable if access to the cloud provider is lost. A client might also become locked-in to a cloud provider if they outsource too much data, as the price to migrate all of the data to a second cloud provider could be prohibitive. Solutions to both these problems appear to be easier. Data can be replicated to several clouds which would solve the availability problem. However, just replicating all the data on a number of clouds presents a serious storage overhead. Fortunately algorithms like eraser codes could be used to reduce that overhead while maintaining the fault tolerance to some of the clouds failing, while hashes could provide integrity checks for

the data replicated.

Can users and companies outsource data to the cloud and have guarantees that their sensitive data is protected, while remaining available and correct whenever they need it? Is it possible to build a system that offers all these three security guarantees (privacy, integrity and availability) that also offers the possibility of executing searches on encrypted data, particularly multi-modal data, while also maintaining performance that allows it to be used in practice? While systems that address some of those concerns exist, none of them try to address all three of those security guarantees while allowing searching on encrypted multi-modal data.

1.3 Objectives and Contributions

The objective of this dissertation is focused on the design, implementation and experimental evaluation of a solution implementing a trustable and privacy-enhanced indexing and searchable multi-cloud encrypted storage backend. The goal of such system is to allow data-storage, indexing and searching operations on documents (objects) that can be composed by text or images.

For our purpose, the designed solution provides the necessary support for multi modal online searching operation on the encrypted documents, always maintained encrypted on the used cloud-services, leveraging on the transparent integration of diverse solutions from multiple cloud-storage providers.

Our solution is based on a middleware system supporting remote storage and multi-modal privacy-enhanced indexing and searching of cloud-based encrypted documents. These documents can be fragmented, and the fragments can be replicated in a dependable multi-cloud repository backend.

Summarizing, we achieved the following results and contributions:

- The design of trustable and privacy-enhanced searchable storage middleware architecture based on a dependable multi-cloud approach;

- The implementation of the proposed solution in a research prototype that can be used for the intended purpose and ready for the possible integration of different applications (via the provided external APIs). The solution is composed by two main macro components: a client-component that will be used as a client-proxy (for local application support), and a middleware service component that can execute in a trusted virtualized docking based appliance, running remotely in a computational cloud. The former will support object indexing, fragmentation and replication, transparently, in the multi-cloud encrypted storage backend;

  - We implemented the integration of two different variants for the multi-cloud storage backend: a replicated multi-cloud data-store leveraged by the Depsky

solution [12], and a replicated multi-cloud in-memory data-store supported by the implementation of the RAM-Cloud solution [13].

  - Among the different components of the middleware solution, we emphasize the implementation of a Java JCA [14] compliant cryptographic-provider library (standardizing its design for the generic use as any other JCE-cryptographic provider), implementing new cryptographic primitives for content-based searchable encryption and multi modal searchable encryption constructions [15].

- We conducted the experimental evaluation of the proposed system, analyzing the validity of the system design model, performance, and data-access latency for searching and retrieving operations. In this evaluation we included the comparative analysis of the two multi-cloud storage backend solutions: Depsky-based and RamCloud-based.

Our results show that the proposal offers the desired security and privacy guarantees and provides efficient privacy-enhanced information retrieval capabilities, without sacrificing precision and recall properties on the supported search operations, if compared with the same operations accessing non-encrypted data stores.

1.4 Document Structure

This document is organized in six chapters. The second chapter is dedicated to current efforts to solve the issues presented in this chapter, from novel encryption schemes to trusted computing and state-of-the-art middleware systems that are similar to the one presented in this thesis. We also compare those middleware systems and see where this thesis improves the current solutions over those already developed middleware systems. On the third chapter we state the adversary model and an initial view of the system model and architecture. On the fourth chapter we detail the implementation details, including the technologies used and the evaluation environment. The fifth chapter includes the description of the test benches and datasets used as well the experimental observations and analysis. The sixth chapter includes the conclusions and future work.

CHAPTER 2

RELATED WORK

Outsourcing data to the cloud while maintaining guarantees of privacy, availability and integrity is an hot topic actively being developed. In this chapter several approaches are discussed that can be used to achieve one or more of these objectives. In Section 2.1 we discuss cloud privacy and how it can be achieved. We discuss cryptographic schemes, presenting mechanisms and tools, oblivious schemes and multi modal searchable techniques. We will then move in Section 2.2 to trustable and secure cloud storage, which tries to address the problems of availability and integrity of data in the cloud. We analyze several state-of-the-art approaches and in Section 2.3 we analyze other approaches like Google's Bigquery Platform, RAMCloud, OpenReplica and solutions that try to look at clouds as a RAID disk. In Section 2.4 we look at the state-of-the-art in trusted computation and in Section 2.5 we do a critical analysis of the analyzed approaches.

2.1 Cloud Privacy

When outsourcing data to the cloud all guarantees of privacy are lost if the data is in plain text. It is necessary then to find an efficient method of encrypting it while not incurring in an excessive overhead for it's use. We look at several approaches that allow privacy to be maintained when data is sent to the cloud and see their benefits and problems. These approaches are secret sharing, threshold signatures, homomorphic encryption SSE, Path ORAM, Multi-Cloud Oblivious Storage and multi modal searchable encryption.

2.1.1 Cryptographic Mechanisms and Tools

2.1.1.1 Secret Sharing

Secret sharing schemes are designed to distribute a secret among several participants. Each participant is given a share of the secret and the secret itself can be reconstructed if enough

shares are joined. This schemes have two essential properties: given any t or more pieces D_i computing D is easily done; with any $t-1$ or less pieces D_i D is undetermined such that all possible values are equally likely.

Secret sharing schemes that do not follow the second property are called imperfect schemes. One example of an imperfect scheme is Blakley Secret Sharing Scheme [16]. This scheme uses an hyperspace in which t planes, out of n, can be used to define a point. That point is the secret and each plane is a secret share. By knowing some of the planes the solution space can be reduced to the intersection of those planes. Shamir Secret Sharing Scheme is an example of a perfect secret sharing scheme [17]. A polynomial p is used to share the secret and t points in the plane of that polynomial are the secret shares. The polynomial is reconstructed by using the points.

2.1.1.2 Threshold Signatures

Digital signatures provide a way to guarantee authenticity and integrity to data and it's widely used with emails and electronic documents. A public key can be used to verify data signed by the private key. This is however limited when multiple parties represent a single entity. In those cases multi signature and threshold signature schemes are used. Multi signature schemes don't have a minimum number of participants and each participant signs the data individually. In threshold signature schemes the participants sign the message using a split key [18]. A split key is distributed among all participants and enough participants are required to generate a valid signature. This ensures that the message was signed by enough parties to represent the entity.

Current threshold signature schemes are based on ElGammal (discrete logarithm problem), RSA (factorization problem), Elliptic Curve ElGammal (elliptic curve discrete problem). They allow any t or more participants to generate a valid signature and the verification of the signature by using the group's public key, without identifying any of the signers.

2.1.1.3 Homomorphic Encryption

Homomorphic encryption schemes have been proposed as a way to execute operations on encrypted data. This attempts to solve the problem that when outsourcing encrypted data to the cloud there is no way to execute operations on it without a big overhead on decrypting and re-encrypting it. There are two groups of homomorphic encryption schemes: partial and full.

Partial schemes, like Paillier [19], allow only some operations. Full schemes allow all operations. Full homomorphic schemes have been proposed by Gentry [20], and Djik et al [21], however they have a very poor performance and retrieving all the data to do all the processing locally and then re-upload it is often less expensive. Due to this, several partial schemes are used instead. Partial schemes however are still much slower when compared to other more traditional cryptographic schemes.

2.1.1.4 Searchable Symmetric Encryption

Searchable symmetric encryption schemes allow an user to search over the encryption data avoiding some of the overhead costs of having to retrieve all the data, decrypt it, do the necessary operations, re-encrypt it and send it back to the cloud. The data is indexed by the client, with the data being encrypted with a probabilistic encryption scheme and the index with a weaker encryption scheme (deterministic or order-preserving).

There are several algorithms available for both text and image searches [11, 15]. On text algorithms a keyword/file pair and search tokens are generated with a secret key, while for image algorithms color histograms, shape descriptors or salient points are used. Security guarantees degrade with every query until a minimum threshold and in most cases they are designed for a one reader/one writer scenario

The first SSE scheme proposed that supports multiple writers/readers [22] encrypts all documents with a different key. To search users send a search token and a delta between their key and the key of documents they have access to. Although each delta is only calculated once, the storage requirements on the cloud increase exponentially the more documents and more users exist limiting scalability and it has a linear-time search. The cryptographic schemes are also slower than traditional symmetric encryption.

2.1.2 Oblivious Schemes

While encrypting data may seem like it's enough to protect data, an attacker might still learn information about it by looking at the access patterns, seeing which data is returned for a query, and which data is more frequently read or written. Oblivious RAM schemes were initially proposed by Goldreich and Ostrovsky [23]. Initially designed to be a scheme of software protection, it was the base for the development of oblivious storage that can be applied to the cloud by hiding the access patterns to files following the same idea that allows to hide the access patterns to memory areas.

2.1.2.1 Path ORAM

Path ORAM, present in 2013 by E. Stefanov, et al [24], hides access patterns to a remote storage by using a small client storage. This is done by continuously re-encrypting and moving data in the storage as it is used. So although an attacker can see the physical accesses to the data they can't know which data is actually being accessed, since which data is in that position on the storage has changed since it's last access. To achieve this the algorithm sees data as blocks of a certain size that are stored in the server in a binary tree.

Each node from the tree is a bucket that can hold several blocks and unused blocks in a node are filled with dummy blocks. The client stores a stash which can hold some blocks and a position map. The stash is usually empty after operations and it's used to hold blocks that overflow during the execution of an operation. The map stores the path in which a given block is and when requesting a block, the whole path will be retrieved. After accessing a

block it's position is changed. Client side storage can be reduced by using recursion and storing the position map in the cloud.

2.1.2.2 Multi-Cloud Oblivious Storage

Presented in 2013 by E. Stefanov and E. Shi [25], this algorithm uses multiple clouds to store data. The objective is to reduce the bandwidth cost of using an ORAM algorithm. The best ORAM algorithms known have a 20X-35X bandwidth cost over accessing the data directly, while with Multi-Cloud Oblivious Storage that cost can be reduced to 2.6X. This is achieved by moving the higher bandwidth cost to be inter-cloud instead of client-server. While it requires a cloud with computational capabilities, it doesn't require secure computation which makes it possible to deploy this algorithm in the current clouds. It is assumed that the clouds are non-colluding and at least one cloud is honest. Although only two clouds are used in the paper, it is possible to use more. The client side storage required is $O(\sqrt{N})$, with N being the number of data blocks outsourced to the clouds.

To prevent incurring in bandwidth cost for the client, the clouds encrypt and shuffle the data between themselves. After the access to some data, one cloud shuffles it, adds an encryption layer and sends it to the other cloud. On the next access the clouds switch roles, so that one cloud sees the access pattern, and the other sees the shuffling. An encryption layer is needed so that one cloud can't recognize a block of data when the other cloud sends them after shuffling. Otherwise it would be possible for a malicious cloud to detect an access patterns when operations are executed over data, even with shuffling. To detect a deviation of the algorithm a commutative checksum-encryption is used, so clouds can monitor each other actions and allow the client to verify the correctness of the protocol.

2.1.3 Multi Modal Searchable Encryption

Multi Modal Searchable Encryption was first proposed in [15, 26] by Ferreira et al. To support searching in encrypted data a new cryptographic algorithm was developed called Distance Preserving Encoding (DPE). The designed DPE scheme has two instantiations to be applied to dense and sparse data (image and text respectively). The scheme preserves a controllable distance function between the ciphertexts. Since some distance between the ciphertext is maintained some information is leaked about similarities between plaintexts. The distance preserved can be configured so that this leakage can be controlled on a application basis. If the distance between ciphertexts is bigger than the defined threshold then nothing is leaked.

Dense data like images are characterized by having an high dimensionality and a non-zero value in all of its dimensions. This means an algorithm must be able to deal with high dimensional feature vectors and keep a controllable distance between them. To achieve this the features vectors are transformed with universal scalar quantization, which preserves distance l_2 between the plaintext features vectors for a distance l_1 between the ciphertext vectors if that distance is less than a configurable parameter t. If the plaintext distance is

bigger than t then the distance between ciphertexts will tend to a constant. Sparse data like text is characterized by having a limited number of non-values, for example, an english document will contain only a small subset of the english vocabulary. This means that to compare two features vectors it's only necessary to compare for equality the non-null values. This means threshold t will be 0 and it will only leak if two keywords are the same.

2.1.4 Discussion

Secret Sharing schemes provide an interesting approach to share encryption keys. However, other symmetric encryption schemes(as presented in...) are more efficient to encrypt the files. Futhermore is always possible to use a key distribution service with those schemes in orthogonal way. Threshold signatures are schemes to authenticate documents signed by several parties. However when data is outsourced to several clouds which are untrusted from a single client, and when there is only one party signing it, there are no advatages in using threshold signatures.

Homomorphic encryption schemes have a considerable ciphertext expansion and are still too slow for a practical environment. Alternatively, SSE schemes are faster, however requiring a lot of client computation and bandwidth. This poses a problem when lightweight and mobile devices are used in the client side. Multi Modal searchable encryption tries to minimize the impact seen in searchable symmetric encryption schemes on the client by moving most computations to the cloud while maintaining privacy.

The approach of Oblivious Ram is interesting to hide access patterns. Although both Path ORAM and Multi-cloud Oblivious Storage incur into a expensive overhead in bandwidth cost while also decreasing the performance of the system. These solutions were used as starting points to provide obfuscation on the access patterns to files on the storage clouds an issue which is not in our objectives.

2.2 Trustable and Secure Cloud Storage Systems

Storing data on the cloud can be cheaper than hosting the data on a company's own server, but if access to the cloud is lost so is access to the files. This can happen for several reasons like connections issues, cloud hardware failure, denial of service attacks (DoS) or malicious administrators corrupting the data. It can also reduce the speed at which files are written or read due to the latency in contacting the cloud servers for operations and create a vendor lock-in problem for the company. In this Section we look at different approaches to make cloud storage more reliable and secure.

2.2.1 Farsite

Published in 2002 by Adya et al. [27], Farsite is a remote storage system that provides privacy, availability and integrity. Files are encrypted with symmetric encryption and replicated across several machines and one way hash functions provide detection for file corruptions. Each machine in the system can have three roles: client, directory group or file host. Files are stored logically in a repository, which is called a namespace. Directory groups are responsible for managing a namespace and file host are machines that host the file's data. Namespaces are managed through a Byzantine-fault-tolerant protocol. To access a file a request is sent to the machines responsible for the namespace where the file is stored. If a machine fails for a long period of time it's functions are migrated to another machine using the replication to regenerate the lost data.

Despise addressing privacy, availability and integrity, Farsite also trys to address scalability issues. It was assumed a maximum of 10^5 machines in the system but a small number of concurrent I/O operations for files. Although it can support large scale of read only operations on the same file, the system is not designed for scalable write operations. The system design is more intended to provide the functionality of a local file system with the benefits of a remote replicated file system with low sharing requirements.

2.2.2 EHR

A privacy preserving Electronic Health Records (EHR) systems was developed by Narayan et al. [28] in 2010. This is achieved by a combination of symmetric and asymmetric encryption, specifically attribute-based encryption (ABE). ABE schemes allow security to be based on a set of attributes. In a ABE system each user's public key has a set of attributes and each ciphertext has a access policy based of those attributes. Decrypting the data is only possible if the private key has the right set of attributes.

File metadata is encrypted with an user's public key and the data itself is encrypted using symmetric encryption. Keys are generated by a trusted (TA) authority that verifies the private key's attributes and public key's are published in a public directory. The ABE scheme used allows users to give access to only parts of the file and to revoke access directly without having to re-encrypt the files or metadata with a different key. It also allows an user or entity to delegate some it's attributes to another user or entity. Searching is supported with the use of a Secure Channel Free Public-Key Encryption with Keyword Search which doesn't reveal keywords or partial matches. Although this system addresses privacy concerns, it doesn't solve the integrity and availability problems. It also requires a central TA to issue or verify the keys of every user which will have access to all encrypted files, it's search capabilities only work on text and doesn't support ranked matches.

2.2.3 Silverline

Silverline is a middleware designed by Puttaswamy et al. in 2011 [29]. It aims at providing data confidentiality for applications using cloud storage to keep a database with minimal changes to the application code. In order to maintain the application performance Silverline leverages the fact that the cloud doesn't need to know exactly what the data being stored represents to be able to perform it's functions. Data that is used in the cloud for computations, like calculating the average age of users, is stored in plain text. Other data, called functionally encryptable is encrypted with a symmetric key. For example a SELECT query for a given user id, the cloud doesn't need to know which user id is been processed, only that the query compares two equal values. Encryption keys for users are generated and managed by the organization responsible for the application. Users can retrieve and store their keys through browsers and HTML5 to reduce network traffic and load on the organization servers.

To set up a Silverline system a training phase must be executed, in which the application perform a set of queries. This queries will determine the minimal key set for users and which data is functionally encryptable. This can be problematic for larger databases in which is hard to get a set of queries that represents real use case of the application. The database will have to be changed so that the encrypted data is stored correctly although those changes are column type changes and not in the database structure and the application will have to be changed to communicate with the middleware instead of with the cloud directly. It doesn't support ranked queries, or keyword searches since it compares the whole ciphertext in the database field and doesn't address availability or integrity.

2.2.4 Depsky

Depsky is a cloud-of-clouds storage middleware proposed by Bessani et al. in 2011 [12]. It addresses confidentiality, availability, integrity and vendor lock-in by using several clouds, symmetric encryption, secret sharing and erasure codes. It supports a byzantine fault tolerant protocol so up to f in $3f+1$ clouds can fail and data remains available. The system is also extendable to support more, or different, clouds which deals with the vendor lock-in problem.

To provide confidentiality file data is encrypted with a randomly generated symmetric key. This key is then shared among all clouds using a Secret Sharing scheme. File data, organized as data units, are replicated in multiple clouds using erasure codes. This scheme allows the storage space used on each cloud to be around 50% of the size of the original data. Depsky also allows the user to use only secret sharing, erasure codes, or none when storing data. The authors also propose a low contention locking mechanism that uses the cloud themselves to manage concurrent writes.

2.2.5 iDataGuard

Proposed in 2008 by Jammalamadaka et al. [30], iDataGuard is a cloud storage middleware that aims to support multiple clouds. It addresses confidentiality and integrity by using PBE and HMACs. Encryption of files uses a symmetric key constructed with a master password given by the client and the object's name. Each file will have a different key to prevent cryptanalysis attacks. Data integrity is achieved by using HMACs constructed by using the file's content and the version number. Since version numbers can't be stored together with the file iDataGuard stores an object in one cloud and the version number in another cloud.

Text and pattern search are supported by an inverted index with all the keywords and a list of documents where they appear, and a q-gram index that maintains all the possible q-grams and a list of keywords that contain them. Both indexes are encrypted and stored in the cloud. When an user wants to perform a search the required index is retrieved and consulted, instead of retrieving all documents. For a large number of files however these indexes can become too big. File system operations are translated into abstract operations that the Service Adapters (SAs) can perform, while SAs translate those abstract operations into operations supported by the cloud provider. These SAs will be cloud specific and independently developed SAs and can be installed into iDataGuard to support more cloud providers.

2.2.6 TSky

TSky is a middleware framework proposed by J. Rodrigues in 2013 [31]. It uses a cloud-of-clouds to store data and ensures confidentiality, availability, integrity and avoids vendor lock-in. TSky can be run locally or as a proxy running on a trustable machine. It uses LSS and Paillier to provide searchability and scoring updates in encrypted data, threshold signatures for data availability and integrity, and symmetric encryption with secret sharing for data confidentiality. The API provided by the framework consists of PUT/GET/REMOVE and LIST operations and several adapters are used to communicate with the clouds in a transparent way to the client.

A master key and reference are generated and stored locally. The master reference is the hash of a cloud object containing the hash of a replica and the cloud in which it's stored. A replica contains all the necessary information to retrieve the data. The master key encrypts the replica's hash and the reference to the cloud where it's stored. To retrieve some data the master reference is used to get the encrypted cloud object associated with it. The master key is used to decrypt this data and obtain the replica's hash and cloud reference to where it's stored the data. Using the replica's hash the data is retrieved, the seed used to generated the symmetric encryption keys is regenerated from the secret shares, the file's data is decrypted and then verified it's authenticity with the threshold signature schemes.

2.2.7 Fairsky

Fairsky is a cloud-of-clouds middleware proposed by A. Strumbudakis in 2013 [32]. It provides confidentiality, integrity, availability and cost analysis of the best clouds to use based on file use. The API provide a transparent interface with a set or regular functions in file systems: PUT, GET, REMOVE, LIST, MKDIR and SEARCH. The connectors provide an abstraction to the heterogeneity of the cloud interfaces so the components in the middleware can use the interface for all clouds.

The index consists of a list of file entries and their fragments list with each fragment having a reference to the cloud where the data is being stored. The data stored in the clouds consists of some metadata like the symmetric key used and the hash of the data. The symmetric key is stored encrypted with asymmetric encryption so it can also be used as means to implement access control by storing it encrypted with the public key of each user who has access to the file.

An application profile configured by the client is used to decide how many replicas to use, how much file fragmentation should be done and the priority of read, write and search operations. In addition to those factors the latency for read and write operations, remaining storage before the pricing model changes and how much does each read and write operation costs, as well as current storage price are used to decide in which cloud to store the data fragments. These last factors are measured while the application is running and can change from one operation to the next. Although Fairsky has a search operation it only supports search over the file's names. It also uses a multiple-reader one-writer model which can limit scalability.

2.2.8 MICS

Mingling Chained Storage Combining Replication and Erasure Coding (MICS) is cloud remote storage system proposed by Y. Tan et al. in 2015 [33]. It offers availability and high performance in trade of some space overhead by using chain replication and erasure codes. Chain replication offers high read performance and tolerates up to $N-1$ cloud failures, with N being the total number of clouds used. The drawback is that chain replication uses raw replication and as such it might not be practical with the storage overhead. Eraser codes are used to reduce that overhead. The system is divided into clients, proxy server clusters (PSC) and object storage clusters (OSC). PSCs maintain a global state of the system and direct the requests to the correct OSC.

MICS was designed to offer high performance and so tries to use the advantages of both chain replication and erasure codes on each operation. Read operations will be directed to the Master Node (MN), which is essentially a copy of the file, so that network traffic is reduced and there is no need to reconstruct the file. There are two types of write operations considered: sequential and random. Sequential writes are initial uploads or overwrite operations. In this case the write is directed to the MN. After the MN stores the file the erasure codes are calculated and pushed to the other clouds. A random write is a update

on a file. In this case the request is directed to the Erasure Coded Chain which uses Parity Logging. Because of the write algorithm there can be problems related to consistency. MICS provides PRAM consistency, so write operations done by a single client will be seen by every client in the order they were made, but there are no guarantees as to which order write operations done by different clients will be seen.

2.2.9 Discussion

Individually, none of the analyzed systems provide a system with the same characteristics as the one proposed in this thesis. Fairsite lacks the search capabilities intended, while also being designed to a smaller scale deployment setting. EHR doesn't address availability or integrity of data, and also trusts all of the data stored to a single central authority, for example a single cloud. Silverline tries to use conventional encryption schemes and so makes some trade-offs when storing data. One of them is that data is not encrypted if it's required for some operation in the cloud. Unfortunately data can be both sensitive and required for operations. Depsky addresses the availability, integrity and privacy of data however doesn't offer any form of searchability on encrypted data which limits it's use. Fairsky, while addressing the main issues of availability, integrity and privacy also lacks search capabilities and only supports a single writer policy, even if writers are modifying different files. MICS only addresses availability and both iDataGuard and TSKy offer searchability but only on text data.

2.3 Other Approaches and Tools

In addition to the previously presented solutions, there are other approaches to the problems of cloud storage. Google's Encrypted BigQuery is one such approach, in which an encrypted database is provided as a service for the client. Other possible approach are cloud-raid systems that treat clouds as raid disks. Encrypted BigQuery [34], RAMCloud [13], Memcached [35], OpenReplica [36], RACS [37], Cloud-RAID [38, 39], NCCloud [40] and InterCloud RAIDer [41] will be presented in this Section.

2.3.1 Erasure Codes

Erasure codes are widely used in storage solutions as way to deal with errors and failures [42]. Unlike data being transmitted and received in real time, data stored can't be retransmitted in case an error occurs that corrupts it creating the necessity for error correction on top of detection. There are several kinds of erasure codes each one presenting advantages and disadvantages. For example, if we use an erasure coding scheme that replicates all data on a second storage, we have the advantage of redundancy and access to the data with a low computational overhead. However, the disadvantage is that such a scheme will require much more storage. More complex erasure codes, like Reed-Solomon codes, can

provide similar redundancy with less overhead on storage. However this approach is more computationally expensive.

The complexity of the required computations depends on the codes and the used parameters. Indeed, erasure codes depend on several parameters, both for redundancy and performance. For the most simple codes k data blocks are used to create m coding blocks using w-bit words. Those three parameters define the fault tolerance, storage overhead and performance. The most common values of w are 1 and 8 (which means a word is a byte). Higher values are possible, however with a significant impact on performance due to the Galois Field arithmetic required for the coding. The fault tolerance depends on both m and the code used. If a code can tolerate and recover from the failure of any m blocks then it's optimal and called a maximum distance separable code (MDS). These codes provide the optimal balance between storage overhead and fault tolerance.

When dealing with a distributed system and objectives as stated in this thesis another factor has to be considered which is the cost of recovery. How to deal with the failure of one of the storage clouds? Although data remains available it means that another cloud can't fail without affecting the availability of the stored data. A direct solution in which all data is retrieved and redistributed is not acceptable due to the computational requirements and network traffic generated. Other solutions can be explored, and erasure codes that minimize this problem, called Regenerating Codes [43], is still an area of active research.

2.3.2 Google's Encrypted BigQuery Platform

Encrypted BigQuery is an extension to BigQuery that alters the client connecting to the database while maintaining performance and being scalable [34]. It works by supporting an extended schema for tables in which an user can choose how to encrypt each column. Several encryptions schemes are supported including homomorphic, probabilistic and some adaptations of more traditional schemes to support searches. This allows the user to query the encrypted data and perform some operations. The encryption key is stored on the client and it's distribution is left to the user. Not all data needs to be encrypted, so non-sensitive data be stored in plaintext to improve performance.

Although Encrypted BigQuery provides availability by being built on top of BigQuery still has several problems to it's adoption. In order for data to be encrypted in a secure way only a subset of BigQuery's operations are supported and it's search capabilities are limited to text. The user also needs to define when making the tables schema which encryption scheme will be used for each column. This takes away some flexibility as if later the application is changed to support a new query to which the encryption used isn't adequate it will force either a performance loss because data will have to be retrieved and the processing done in the client or a database re-encryption.

2.3.3 RAMCloud

RAMCloud is a solution based on in-memory storage to improve the performance of cloud based web applications [13]. While applications running on a single machine can often store the required data to work on DRAM and access it in $50ns - 100ns$, web applications have to store their data on separate servers causing access times of $0.2ms - 10ms$. RAMCloud uses a key-value data model and a log structured storage. Data is organized in tables, which might be split over multiple servers in tablets. Objects in the log are referenced by entries in a hash table stored in the server.

There are three types of servers on RAMCloud: masters, replicas and coordinator. Although a single server can act as both master and replica, replicas must be on a different server than the master responsible for the data. Data written to RAMCloud will be assigned to a master server which will keep it on DRAM. Replicas will store that data on disk to provide persistence. Coordinators direct clients to the servers responsible for the data requested.

2.3.4 Memcached

Memcached is a solution designed by Fitzpatrick to improve the caching in web servers [35]. The intention was to cache the objects needed to generate dynamic web pages, but not the pages themselves as that would lead to some page objects being cached more than once and being requested less often. A regular caching solution would be able to cache the objects instead of the pages. However that would not solve the problem of an object being cached more than once as multiple servers could need it but they couldn't access each others cache. Memcached allows servers to access a single cache that is distributed among those servers.

Cache entries are organized in a two layer hash table. The first layer indicates in which server the entry is located, while the second one accesses the object in the server. Memcached server instances are independent of each other and it's the client library that decides in which server a certain key is stored. Several client libraries are available and the protocols to communicate with the Memcached servers are available allowing the development of specific client libraries for applications. This creates a single cache that has the size of the DRAM available in all the server running Memcached combined, instead of several smaller instances. Each Memcached instance can also have different space available for caching allowing an easier deployment in different machines.

2.3.5 OpenReplica

OpenReplica is an object-oriented coordination service [36]. It operates on objects that represent state machines provided by the user and turns them into fault-tolerant replicated state machines. Clients don't need to be aware of the replication and don't need to instantiate the object itself. Operations are called on a proxy object created by OpenReplica that provides an interface similar to the one of the object provided. Paxos is used for coordination and

operations are executed in a strict order so developers don't need to concern themselves with locks or idempotent operations that can be executed more than once due to failures or delays in the network. Replicas are instantiated in several servers and name servers direct clients to those servers. To provide at most once semantics a command log is kept. The order of the commands in the log is decided by Paxos.

2.3.6 Cloud-RAID Solutions

RAID systems provide integrity and availability by using striping in which data is split over all disks and by using redundancy in which the same data is written in more than one place. Redundancy can be implemented with raw data replication, parity bits or erasure codes and provides availability in case some of the disks fails. How many fails it supports depends on the RAID level [44]. Some systems that approach clouds as RAID disks will be presented on this section.

2.3.6.1 RACS

Redundant Array of Cloud Storage (RACS) was presented in 2010 by H. Abu-Libdeh et al [37]. It was designed as a proxy that writes data to clouds with minimal changes to applications by using a subset of operations from Amazon S3 REST API. This system provides availability, integrity and deals with vendor lock-in by using multiple clouds as storage. Like RAID approaches on disks, RACS uses a mix of striping and redundancy with eraser codes to write data and maintain availability without incurring in the price of full replication.

Clients that are modified to use all capabilities of RACS can also give some policies hints on requests so that RACS can choose repositories better suited to that operation for example based on latency, load balancing or pricing model. To avoid becoming a bottleneck RACS can work in a distributed mode in which several RACS proxies are connected to the same set of cloud repositories and maintain the same state for user authentications. Zookeeper is used to coordinate all the proxies and to maintain a one-writer multiple-readers policy for each key in a bucket.

2.3.6.2 Cloud-Raid

Cloud-RAID is a system proposed by Schnjakin et al [38, 39]. It provides privacy, availability, integrity and avoids vendor lock-in by adopting some RAID techniques. Each cloud is seen as a RAID disk and symmetric encryption and eraser codes are used to store files in the clouds. Cloud-RAID also tries to dynamically store files in the best cloud providers according to user criteria, for example storage price or quality of service expectations. Cauchy-Reed-Solomon algorithm is used to split files and after encoding the fragments are encrypted with AES. The resource management module has a database back end and keeps track of the information necessary to retrieve the files like encryption keys, original file size and hash and eraser code algorithms.

Cloud providers are abstracted into storage repositories and only support six operations: create a container, write a data object, list all data objects, delete a data object and retrieve the hash of a data object. Data objects represent the file fragments and are stored in containers. To keep track of cloud providers services there is a reputation object which stores information about past performance (storage price and response time for example). This object is then used to choose which cloud providers to use. Although providing privacy Cloud-RAID doesn't support any operations on encrypted data. Encryption keys are also all stored locally in a database which can become a single point of failure.

2.3.6.3 NCCloud

NCCloud is a system proposed in 2013 by H. Chen [40] that aims to provide a RAID6 storage system over multi-cloud storage. It improves on RAID6 by using functional minimum-storage regenerating codes (FMSR) which generate less network traffic when recovering data but doesn't offer any privacy or integrity. Using the NCCloud system with four clouds the file is split into four chunks and then generate eight chunks obtained by different linear combinations of the original four chunks. Two of the eight chunks are stored in each cloud provider. When a cloud provider fails its two chunks can be recovered by retrieving any three of the remaining six chunks. Although NCCloud deals with availability better than other systems it doesn't provide integrity or privacy for data. It also only supports file upload, file download and file recovery operations. The system evaluation for availability purposes was conducted in a four cloud scenario where it's possible to tolerate up to two failures instead of one.

2.3.6.4 InterCloud RAIDer

InterCloud RAIDer is a system proposed in 2014 by C. H. Weng and A. Datta [41]. It provides privacy, integrity and availability however it uses different techniques than the ones used in the systems presented previously. It uses chunk-based deduplication to split the files over the clouds, non-systematic eraser codes for redundancy and provable data possession (PDP) for integrity.

The eraser code chosen is a homomorphic self-repairing erasure code (HSRC). Unlike systematic eraser codes, non-systematic eraser codes don't have any of the original data in the encoded blocks, so unless a cloud provider gets access to enough encoded blocks from the same file to reconstruct it completely it won't be able to retrieve any of the original data. After the blocks chunks are encoded with the HSRC they are uploaded to the cloud.

Although it can be generalized to a multi cloud file system with a predefined interface this system was designed with private users in mind and as such is not very scalable and the concurrency of multiple users accessing the same file for reading or writing is not addressed. The index scheme was also not optimized for a very large quantity of data stored.

2.3.7 Discussion

Google's Encrypted BigQuery is an interesting approach but it's geared towards databases and big data analytics. It also has some limitations on the operations allowed which the programmer has to take into account when deciding the database schema. The wrong schema can be very costly in terms of performance and bandwidth costs and it's not always easy to predict which queries will be made in the future. RACS doesn't provide privacy and offers only two connectors for existing cloud providers and a connector for a network file system. Cloud-RAID provides privacy, availability, integrity and avoids the vendor lock-in while also adjusting dynamically in which clouds should files be stored to better suit the user's needs. However there are no operations on encrypted data and all encryption keys are stored locally in a database leaving key distribution completely to the user. NCCloud translate a RAID6 architecture almost directly to cloud storage and then makes some improvements to reduce network traffic to optimize the cost of recovery in case of failure. However it lacks privacy and integrity. InterCloud RAIDer provides availability, integrity and some weak privacy with non systematic eraser codes. If an attacker gets access to enough chunks of a file they will be able to reconstruct it, which would not be the case with the help of an encryption scheme. It also makes some improvements on storage overhead. It is however more geared towards private user with little to moderate amounts of outsourced data. RAMCloud provides an interest solution which tries to minimize latency. Although the results presented in the paper look promising it's a solution designed to be deployed in a data center. This raises the problem of vendor lock-in. Also since replication is all the same data center availability might be affected in case of data center failure, which would not be a problem on other replication services. When deploying it with servers on different data centers the performance gains might not be as good. OpenReplica also provides an interesting approach to replication and the lineralization of operations by using Paxos to guarantee strong consistency. However the use of Paxos for every request can become a bottleneck if too many client proxies are used.

Although the different approaches presented before are interesting contributions in different areas, they are in general aimed at solving different issues when compared with the thesis objectives. Google's Encrypted BigQuery addresses to solve privacy in databases for big data analytics, RACS addresses availability and vendor lock-in problems, Cloud-RAID addresses dynamic placement of files in the cloud while keeping data privacy, integrity and availability, NCCloud focus on the recovery cost if a cloud fails permanently and InterCloud RAIDer addresses cloud storage for a single user. None of these systems, with the exception of Encrypted BigQuery, address data searching capabilities in their implementations. Encrypted BigQuery can do some searches over text data, not providing multi modal searching facilities.

2.4 Trusted Computing

Trusted Computing refers to technologies to implement trust computing bases, by ensuring the integrity of attested operations performed to trusted anchors. In this Section we will look at the state of the art implementations, their architectures, what they accomplish, and their disadvantages. This support stands for attestation capabilities at load and execution time avoiding for the damage of possible malicious code injections. Then we will move to some implementations of TEEs starting with On-Board Credentials (ObC) [45], Open-TEE [46], Trusted Language Runtime (TLR) [47], fTPM [48] and finally cTPM [49].

2.4.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) [50] is usually provided by hardware to create an isolated execution environment from the normal execution environment applications regularly use, in some cases called Rich Execution Environment (REE). A TEE provides a place for applications to execute sensitive operations as well as provide secure storage and guaranteeing the integrity of the software loaded. The hardware architecture can be implemented with two chips, one that provides the REE, and a weaker processor for the TEE. One example of this architecture are TPMs. The other architecture shares some hardware components for both TEE and REE. The processor can execute in two modes, the secure and normal worlds, for example the ARM TrustZone. Access to peripherals and memory areas is restricted by in which mode is the access executed. Some implementations can use virtualization to support several TEEs at the same time and give each application it's own TEE. TEEs have less functionalities than the REE and although this might restrict what an application can execute in the secure world the Trusted Computing Base (TCB) is smaller and it's easier to audit.

A TEE usually provides five services: boot integrity, secure storage, isolated execution, device identification and attestation/provisioning. Boot integrity can be achieved by using either secure boot in which the boot fails if any component fails its verification or authenticated boot in which the hashes of the components loaded are stored and can be provided if requested. Secure storage is provided by storing a device key and cryptographic functions that are protected from physical tampering and can only be accessed by authorized code. Isolated execution can be implemented by using two execution modes (the normal and the secure world) and having an entry point to move from one mode to another. The normal world, will be responsible for the REE where applications execute while the secure world will be responsible for the TEE for sensitive operations. Device identification can be implemented with a identifier stored in the TEE. Remote attestation can be implemented by using the hashes stored during the boot signed with a device certificate. Provisioning can be achieved by using the public key of the device certificate to encrypt the data being sent to the TEE.

2.4.2 TPM

A TPM (Trusted Platform Module) is an hardware module deployed in the motherboard of a computer, smart card or integrated with the processor that provides the functionalities needed for trusted computing. It is composed of several components as shown in 2.1: I/O, cryptographic co-processor, key generation, HMAC engine, random number generator, SHA-1 engine, power detection, opt-in, execution engine, nonvolatile memory and volatile memory [51].

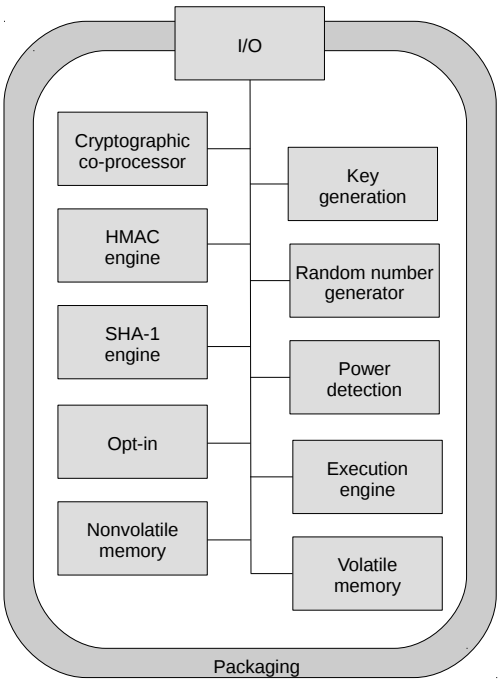


Figure 2.1: TPM components. Adapted from Computer Security: Principles and Practice [51]

The TPM works with the other hardware/software components by providing authenticated boot, certification and encryption services. This allow applications, including the OS itself, to verify that the OS loaded is correct and to securely encrypt data being processed in the machine. To provide authenticated boot the TPM checks the integrity and validity of the code being executed during each stage of booting using digital signatures. A tamper-proof log is also kept of the code loaded so the TPM can check at any time the version and modules of the OS that are running. After booting more trusted applications and/or hardware modules can be loaded by using an approved list. The certification service allows a TPM to certify the system state, based on the authenticated boot service, to other applications. The application requesting the certification sends a nonce along the request and the TPM signs the current system state concatenated with the nonce. This signature is made with a private key that only the TPM has access to, which allows the application to trust that the certification is valid and is not a replay from a previous certification.

This certification service can be used for remote attestation allowing an application that

is executing on a remote machine to attest the integrity of system and applications loaded on the machine running the TPM. When used in the provisioning boot mode this can be used for client applications to attest the integrity of the code loaded on the server. This is done by having the TPM verify the first code executed, usually the BIOS and storing its hash in a Platform Configuration Register (PCR). The BIOS will then verify the next executed code extending the current PCR value. Extending a PCR consists of concatenating the current value with the hash of the new code, and creating an hash of the combined individual hashes. This allows the verification of an arbitrary large stack of software without running out of PCRs, which are limited. Since the TPM is trustable all the system stack can be attested. The client application can generate a nonce and request a quote to the TPM which contains an hash of the values of the PCRs with the nonce signed by a Attestation Identity Key (AIK). An AIK is a asymmetric encryption key registered to the TPM with the public key known to the client. Once the client receives the quote it can verify the signature confirming its validity as well as freshness by comparing the nonce in the quote. It can then compare the hash of the PCRs with known values and decide to trust the system state or not. The known values for the client should be determined ahead of time, for example creating an hash of the PCRs values in a trusted environment. The client can verify the state of the whole system which allows the detection of attacks to system software as well as the server software that the client wants to interact with.

The encryption service allows data to be encrypted and decrypted only when the machine is in a certain state. To achieve this the TPM has a master secret key unique to the machine, which is never shared with any application. With that master secret key several other secret keys are generated for each state. The application requests a secret key from the TPM for the current state and uses it to encrypt the data. An encrypted version of the key with information about the machine state required for the decryption is also provided for the application to keep, while the application is expected to discard the plain text key when it no longer needs it for the encryption. To decrypt the data the application requests the TPM to decrypt the encrypted key that was provided during the encryption. If the machine state is the correct one the key will be decrypted and the application can decrypt the data.

2.4.2.1 ARM TrustZone

ARM TrustZone is an example of a TEE architecture which uses shared components for REE and TEE and splits the two by using two execution modes, secure world and normal world [45, 50]. The processor, ROM, some RAM and peripheral and interrupt controllers are connected by a chip bus. The remaining device components like main RAM memory, storage and antennas are connected with a off chip device bus. Usually the chip components can only be accessed in the secure world mode, while the off chip components can be split between the normal and secure worlds. A status flag in both the on chip and off chip buses indicates in which mode the processor is executing. The ROM contains cryptographic functions, device keys, root trust and device base identity while the on chip RAM is used for runtime isolated

execution of the TEE.

While in the normal world RAM zones assigned to the secure world can't be accessed, but from the secure world areas assigned to both the normal and secure world can be accessed. Isolated execution is provided to applications through a TrustZone library and hardware driver. While in the TEE applications execute on top of a minimal runtime environment which provides an API for applications on the TEE to communicate with applications on the REE and access TEE functions like cryptographic primitives.

2.4.2.2 TrustLite

TrustLite is a TEE designed for resource constrained and low cost devices [46, 50]. Instead of using a Memory Management Unit it uses a Execution Aware Memory Protection Unit (EA-MPU), an extension to Memory Protection Unit. The EA-MPU controls accesses to memory zones based on the current program counter. This provides a fine-grained access control to memory areas. It also provides secure exception engine which protects secure memory even in case of hardware or software interrupts. This provides a lot of flexibility for low cost devices as the EA-MPU is programmable. A device could be programmed to provide exclusive access to a certain application, allows running several parallel secure tasks and shared memory between applications. TrustLite also uses a secure boot loader to initialize the EA-MPU.

2.4.2.3 Intel SGX

Intel Software Guard Extensions is a set of new instructions for Intel CPU architecture [46, 50]. In this architecture the TEE is called a enclave. Enclaves are instantiated inside applications running in the REE and refers to the protected memory area inside the processes virtual address space and the SGX control data structures. Accesses to protected memory areas from outside the application are prevented by the hardware. This restriction is applied to any software, even the OS or BIOS. Data in a enclave is encrypted while stored in the RAM and is decrypted by a hardware unit when being moved to the CPU. Similarly data leaving the CPU is encrypted by the same hardware unit.

While in the enclave the process can access any memory zone on it's own enclave, but not enclaves from other processes. System calls from inside the enclave are prevented and accesses to enclave memory zones outside of the enclave are treated as accesses to non-existent memory. For attestation and provisioning there are two other instructions: EREPORT and EGETKEY. EREPORT provides a encrypted evidence structure and EGETKEY gives an enclave access to the keys used. A special enclave is used for signing the evidence structures with a device specific key.

2.4.3 Software Implementations

Several implementations have been developed on top of the architectures mentioned in the previous Section to address some of the problems that aren't solved by them. In this Section

we will analyze some of those implementations and what they can do.

2.4.3.1 On-board Credentials

On-board Credentials (ObC) was developed at Nokia Research Center [45]. It provides an API that developers can use to create applications that use ARM TrustZone and in particular provides a open provisioning model. While using the ARM TrustZone applications need to be trusted by the manufacturer or other trusted third party, like the OS developer, ObC allows applications to run on the TEE without needing the permission of those trusted third parties, but it still needs the permission of the user. Applications are split into a trusted part that runs in the TEE and a second part that runs in the REE. A byte code assembler or Basic can be used to create applications to run on the ObC interpreter.

The ObC interpreter is a VM that runs on the TEE of the TrustZone. The interpreter provides a runtime environment with cryptographic functions, string and array manipulation, sealing and I/O functionalities. Several applications can run in the interpreter and the state of each one is encrypted and stored when it's not being used. To provide the open provisioning model a device and a manufacturer public keys are used. By using those public keys a provisioner sends a secret key that can be used in a security domain. Trusted applications belong to a specific security domain and data from different security domains is encrypted with different keys.

2.4.3.2 Open-TEE

Open-TEE is a virtual TEE designed to simplify the development of applications that use the TEE capabilities [46]. It uses a Manager process that provides services to the TEE runtime and a Launcher process that creates instances of trusted applications (TAs) and pre-loads the TEE Core API shared library. Each TA process is then split in two threads: an I/O thread responsible for communication with the Manager process and a thread responsible for the TA logic. Open-TEE provides an API that conforms to the GlobalPlatform (GP) standard and is implemented on top of TrustLite and Intel SGX.

2.4.3.3 Trusted Language Runtime

Trusted Language Runtime is a TEE developed on top of ARM TrustZone [47]. It's based on .NET Microframework to provide an easier programming environment while having a smaller TCB than using a regular OS. Applications are split into a untrusted component that runs in the normal world and a trusted component that runs in the secure world. The trusted component doesn't have access to any peripherals or I/O and must rely on the untrusted component for those.

The trusted component of an application is developed in a trustlet. The trustlet defines the interface of the trusted component and which data may cross from the untrusted environment to the trusted one or vice versa. Each trustlet will run inside a trustbox, which is a isolated runtime environment on the trusted environment. Other applications, including the

OS, can't change or inspect any data inside a trustbox which provides privacy and integrity. To provide secure communication channels TLR makes use a public key pair to identify the device and a seal/unseal function. The seal function binds data to a trustlet and a device, encrypting it. The decryption must be done inside a trustbox that is running the specific trustlet and on the right device. While the device identity is provided by the private key of the key pair, the trustlet identity is provide by a secure hash of its code. This allows states to persist across reboots and lets remote applications send data to a specific trustlet.

2.4.3.4 fTPM

fTPM is a firmware based implementation of TPM 2.0 [48]. It runs on a TEE developed on top of ARM TrustZone that consists of a monitor, a dispatcher and a runtime. fTPM itself runs on the runtime, although that runtime can also run other services and a single interface is provided to programs in the normal world through shared memory. Secure storage is provided by a replay protected memory block (RPMB) present in embedded Multi-Media Controllers (eMMCs). The RPMB offers authenticated writes in which a HMAC is stored along with the data and a 32-bit counter is increased. That counter is monotonic and when it reaches it's maximum value that data can't be overwritten anymore. Authenticated reads are issued with nonces to guarantee that the value being read and returned is correct and not an old one.

The TEE monitor is responsible for context switches between secure and normal world. It provides full context switches in which the processor's registers for the current world are saved, and then the registers of the other world are restored, and lightweight switches in which no state of the world is saved. The TEE Dispatcher directs requests to the secure world to the correct service. The TEE runtime provides an interface to secure devices and dynamic memory allocation.

The TPM 2.0 specification requires a secure clock with at least millisecond granularity and a secure random number generator. TrustZone doesn't provide a secure clock but most operations that require a secure clock can be adapted to work on TrustZone on Windows or Chrome devices. The value of the clock is stored periodically to a persistent storage and can be used to measure time intervals. Storing the value of the clock guarantees that the clock value will never roll back to a previous time than the one stored and as such can be used to provide semantics of the type "refuse service for the next x seconds".

2.4.3.5 cTPM

Cloud-TPM (cTPM) is an extension to TPM 2.0 that allows cross-device functionalities [49]. It adds the ability to share a seed with the cloud and access cloud storage. The seed is introduced in the cTPM during manufacture and it's initially used to create a asymmetric root key, cloud root key (CRK) and a symmetric communication key, cloud communication key (CCK). The generation of those keys is done both at the device and in the cloud, as cTPM design assumes that the seed is securely shared with the cloud by external means (for

example at manufacture time, as the interface to add a seed to the device is not available to regular users). The CRK is used to encrypt all data stored in the cloud, while the CCK is used to encrypt all data exchanged with the cloud.

cTPM can't directly communicate with the cloud and must rely on the OS for that. Since the OS can be compromised asynchronous functions were introduced so the cTPM doesn't block for long periods of time when dealing with the cloud storage. This is done by having the cTPM send the data encrypted to the OS, which will then send it to the cloud. The cloud sends back an encrypted response to the OS, which will then be delivered to the cTPM. Commands that don't need to the use cloud storage are still synchronous. Asynchronous commands require the cTPM to store their information on memory. To prevent filling up all the memory if a OS is compromised and refuses network access it was introduced a global route timeout (GRT). The GRT keeps tracks of how long a asynchronous request has been in memory and after a period of time removes it.

Cloud storage is implemented as a key-value store with a index. cTPM also implements a local cache in which entries have a time to live (TTL). Once the TTL is over the entry is deleted from memory. There is also an additional field that stores when was the entry stored in memory. A secure clock is also implemented by using the remote storage. The remote clock is seen as an entry in an index. When trying to read the clock the cTPM issues a read on that index entry, which is subjected to a stricter timeout than regular reads, and then caches the value on the memory. Current time can be calculated by the TPM timer in relation to when the cache was stored. Since there can be a drift on the timer the clock is periodically re-synchronized.

2.4.4 Discussion

There has been a lot of work on going (hot) work in developing TEEs, and most mobile devices (smartphones or tablets) already have the hardware requirements, however most applications available to users don't make use of them. This is because the hardware implementations don't provide a standardized interface and are mostly used by the manufacturers. The lack of standardization interfaces is a serious block to the development of TEE-aware applications as a developer would have to make applications adapted to several interfaces instead of a single one. Hardware implementations also often lack properties required by most applications like a secure clock or secure storage. Several works have been done in an effort to improve TEEs and make them more available for developers but they still have their own disadvantages.

One of the main disadvantages of ARM TrustZone is that it uses a single processor to provide both execution runtimes and lacks virtualization. This becomes visible in ObC as the context switches cause a lot of overhead for the system. Manufacturers often use a less powerful processor for the TEE if the processors for both execution environments are separated. Although ObC is using a more powerful processor, it's performance it's similar to the performance of using a separate and less powerful processor like a smart card. On TLR

the performance issue is also visible. In order to reduce the TCB TLR also doesn't provide access to peripherals which can be limiting on some applications. fTPM had to make several design compromises in order to be deployed in TrustZone. Because of the lack of a secure clock some semantics on TPM2.0 were weakened, for example not allowing time-bounded constrained permissions, and it can't distinguish between a legitimate reboot or shutdown from a brute force attack while booting. Because of that, if the user shutdowns the device, or for some reason it happens, the device will be locked for a period of time before the user can attempt to boot again. cTPM on the other hand assumes that the cloud is trustworthy and only the client device can be compromised. As such, seeds and keys can be stored in the cloud where they can be seen by a malicious administrator. Open-TEE lacks a secure clock, as it's build on top of TEEs that don't have it themselves.

Most TEE solutions are based on mobile hardware and processors as manufacturers need some properties given by TEEs. Although desktop solutions are also available they aren't much more available to users than their mobile counterparts. Cloud based TEEs are also not currently available and the adoption of such solutions incur in an extra expense for cloud providers and possibly to clients. Finally there is a lack of standardizations for generalized implementations by hardware manufacturers.

2.5 Critical Analysis

As seen by the previous Sections, the outsourcing of data to the cloud and the several problems that arise from doing so is an area of active investigation. Although several systems follow a related approach to the one taken on this thesis, they can be improved upon, addressing requirements not covered by those solutions. iDataGuard [30] and TSky [31] are proposals particularly related to the main dissertation contributions. Both these systems provide privacy, integrity, availability and text search capabilities. Text search only is however very limited when we consider that clouds are used to store multi-modal data and multi-modal operations. Other systems analyzed that follow key-value data store models don't provide search capabilities and as such are even more limited (ex. Depsky [12], Farsite [27], Fairsky [32]). In our system model design (approached in chapter 3) we used both in-memory cloud storage backends (leveraged from the RAMCloud solution [13]) and disk based storage backends (leveraged from Depsky [12]), comparing the trade-offs in both. There are also other approaches like Silverline [29] and Google's Encrypted BigQuery [34] geared towards databases instead of key-value stores. However they also have serious limitations: Silverline resorts to not encrypt data that the cloud needs to perform queries, while Encrypted BigQuery limits the possible queries using partial homomorphic schemes that only support text processing.

With the outsourcing of the heaviest computations to the cloud we could improve on trustability requirements in designing the client appliance. By using a TCB that provides secure storage, isolated execution, secure random number generation and integrity loading we can provide a security solution defending from attackers intending to compromise the

client device. The integrity of the client appliance would be attested at loading time, while cryptographic keys would be securely generated and stored. Cryptographic functions would run without danger of memory inspection. To materialize such security requirements we designed our trustability support leveraging from fTPM abstractions [48]. Indeed, this solution provides all the required functions and allows our appliance to be securely deployed on top of TPM2 functionality implemented by ARM TrustZone enabled mobile devices.

Cache is generally used to reduce the time an operation takes or to prevent the server having to process several requests that have the same result. Our approach leverages cache to avoid several requests to the backend to retrieve object fragments and it's reconstruction. This way if an object is requested several times the middleware only has to reconstruct it once reducing the processing and network traffic between the middleware and the backend. Although any cache system could achieve this, Memcached was chosen for several reasons. It was already deployed in several known websites with proofs of its performance and more importantly it is adaptable to possible extensions to this work. By having the cache as a separate server there is the option to deploy it in the same machine as the middleware or move it to another machine (in this case the extra latency would have to be taken in account). Furthermore, because it uses a distributed caching mechanism it makes it easier to adapt if an extension deploying more middleware servers is implemented.

CHAPTER 3

System Model and Architecture

As introduced before, the objective of the dissertation is the design, implementation and experimental evaluation of a middleware system, providing privacy-preserving search and data storage facilities using outsourced cloud-storage backends. The solution combines the support for storage, indexing, searching and retrieval operations for multi modal data maintained always encrypted in the cloud-provided backends. In order to combine the privacy guarantees with other dependability criteria, the system is designed to integrate a multi-cloud storage environment, transparently integrated in our middleware solution as a cloud-of-clouds storage model. In this chapter we present the system model and architecture of the proposed solution. Initially we present an overview of the system model (Section 3.1) and the adversary model definition related to the system design (Section 3.2). Then, we describe (Section 3.3) the system model and architecture in more detail, namely the main architectural components , the cryptographic support for the provided security services and the remaining processing modules of the middleware solution, including the integration support for two variants of multi-cloud provided storage backend services. The two variants are leveraged by two different solutions: Despsky [12] and RamCloud [13].

3.1 System Model Overview

Our system model is composed of three main blocks in which different components are deployed, the cloud storage backend, the middleware server and the client proxy as represented in figure 3.1. On more powerful client devices like a desktop computer the middleware server can be deployed locally together with the client proxy.

The client proxy will be responsible for the extraction of the features required for indexing and searching as well as the encryption of the data. A cryptographic provider will provide an API based on current implemented cryptographic providers. An in-memory cache

for retrieved files is also supported and can be configured or disabled based on the client device capabilities.

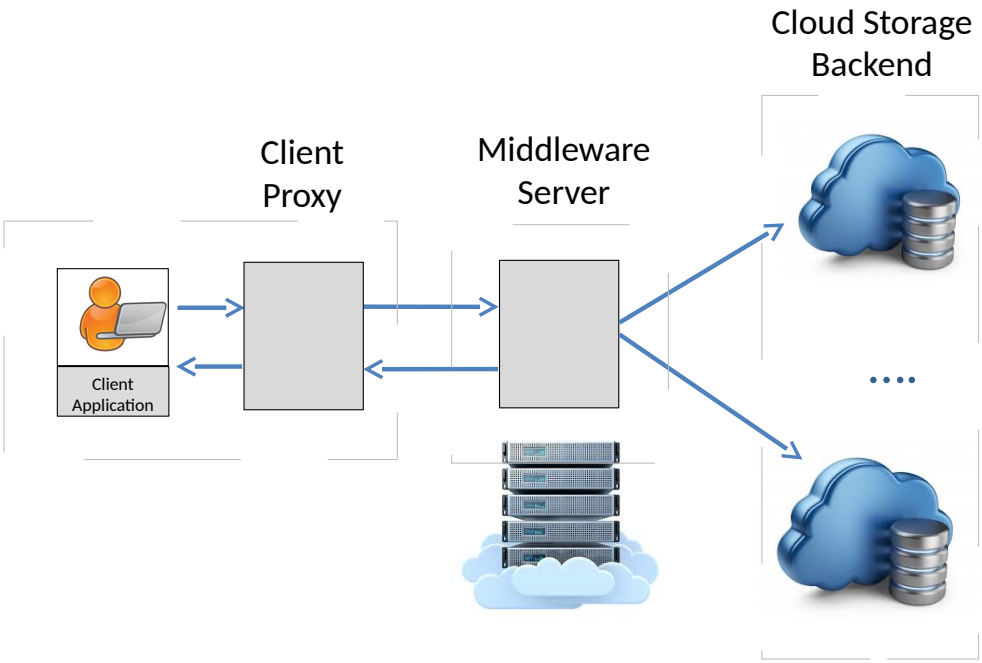


Figure 3.1: System Model

Training and indexing operations which are computationally expensive are done in the middleware server reducing the impact on the client device which is particularly useful for client devices with resource constraints or with less powerful CPUs. Additionally the middleware server will also have a replication module, an in memory cache and several cloud connectors. The replication module will be responsible for the fragmentation and integrity checking of the data. The use of cache on the server and on the client is used to minimize both the latency and the computational overhead. While the server cache will minimize the impact of retrieving the object fragments, the reconstruction and verification of the client cache will minimize the network traffic generated by the client requesting the same object several times. This not only reduces the time the client takes to return a specific object back to the user but also the load on the server.

Two different kinds of storage backends are available, one using RamCloud and another using Depsky. Depending on which one is chosen different clouds have to be used for the storage. In the case of RamCloud we need computational clouds and the cloud connector on the cloud server appliance will be a RamCloud client. In the case of Depsky we only need to use storage clouds and the cloud connectors will use the API provided by each cloud provider.

3.2 Adversary Model

Each of the main blocks in our system has a distinct adversary model. In this Section we will characterize in detail each of these adversary models, indicating what they can and can not. These adversary models are based on the adversary models presented in some of the composing parts leveraged for the development, however they were expanded as our system can deal with more adversary parts than each of the single components.

3.2.1 Storage Backend

The adversary model for the storage backend is the most extensive from the three adversary models presented. It is based from the adversary models presented in CryptDB [52] and Depsky [12]. CryptDB presents two threats: DBMS server compromise and arbitrary threats. While in our system we use a key-value store as opposed to a database approach, we can adapt those adversary models.

The first threat, DBMS server compromise, deals with a honest but curious database system administrator, or an external attacker, with full access to the DBMS and the data stored, both in disk and in RAM. Both the system administrator and the external attacker are interested in seeing which data is stored, but do not modify it or delete it. They also don't change the normal operation of the server. In our system we treat the storage backend as the equivalent of the DBMS considered in CryptDB and fully support this threat. The second threat, arbitrary threats, assumes that all components of CryptDB can be arbitrarily compromised, and guarantees privacy for data encrypted by keys belonging to users that are not logged in during the attack. In this threat model attackers can modify some of the data since attackers would have access to the encryption keys of logged in users. We assume this adversary model only for the storage backend component, but expand on it with Depsky adversary model. We use four storage servers and allow an attacker to create, modify or delete any data in a compromised server, or even arbitrarily change the server behavior, as long as the data in the other three servers remain integer and available. We also assume the possibility of DoS/DDoS to one of the four storage servers.

We deal with this adversary model by using encryption to protect the confidentiality of the stored data, Reed-Solomon codes to fragment the data over the several storage servers and digital signatures to guarantee authenticity and integrity. As long the data in three of the servers is available with integrity guarantees it is possible to reconstruct the original data.

3.2.2 Middleware Server

For the middleware server we assume an adversary model with limited capabilities compared to the one presented for the storage backend. It is neither vulnerable to DoS/DDoS nor code injection at runtime, however an attacker, both external or a system administrator, can inspect, but not modify, all the data being processed. Furthermore, they have access to

all of the disk contents and can modify the server code in storage with the intent to inspect the data processed or store it in a second storage in addition to the regular one, effectively having persistent access to all data processed after the modification.

We make use of client-side encryption to allow an attacker to inspect all the data in the middleware server and use a TPM module to authenticate the code being executed by the server to the client. We are assuming here that a TPM module is present in the cloud. Although in reality that is often not the case, TEEs are the subject of hot on-going research and our system is prepared to make use of it when it becomes more widely available.

3.2.3 Client Proxy

We assume the client proxy to be trusted. Attackers don't have any access to it's memory or disk contents, nor they can do code injections to alter the normal behavior of the proxy. With the use of TPM 2.0 we could extend this adversary model to allow attackers to have access to the disk contents and even modify them. This could be solved then by authenticating the client proxy code with the TPM module as the proxy initiates, and using the encryption capabilities of the TPM to encrypt the proxy encryption keys. Those encrypted keys could then be stored in disk or in the cloud, while the key used by the TPM would be safely stored in the secure storage, accessible only by the TPM itself.

3.2.4 Generic Adversarial Conditions

In addition to the previous adversary models we also make some assumptions about their interactions. We assume non-colluding adversaries and independent failures. This means an attacker looking to exploit one of the components of our system won't use an attack made by a different attacker together with is own nor will be caused by it.

3.3 System Model and Software Architecture

In this Section we will describe in detail each of the main blocks of our system as well as their components. The main blocks are the client proxy, the middleware server and the storage backend. Each of these components is deployed in a different location, although the middleware server could be deployed together with the client proxy on more powerful client devices, such as a desktop computer. However, since the middleware server isn't replicated, this would either allow only one client, or the desktop computer would have to act as a server for the other clients, which could impact their quality of service. Previously, in Section 3.2 we present an overview of the components that make each block. In next sub-sections we will discuss those components in more detail.

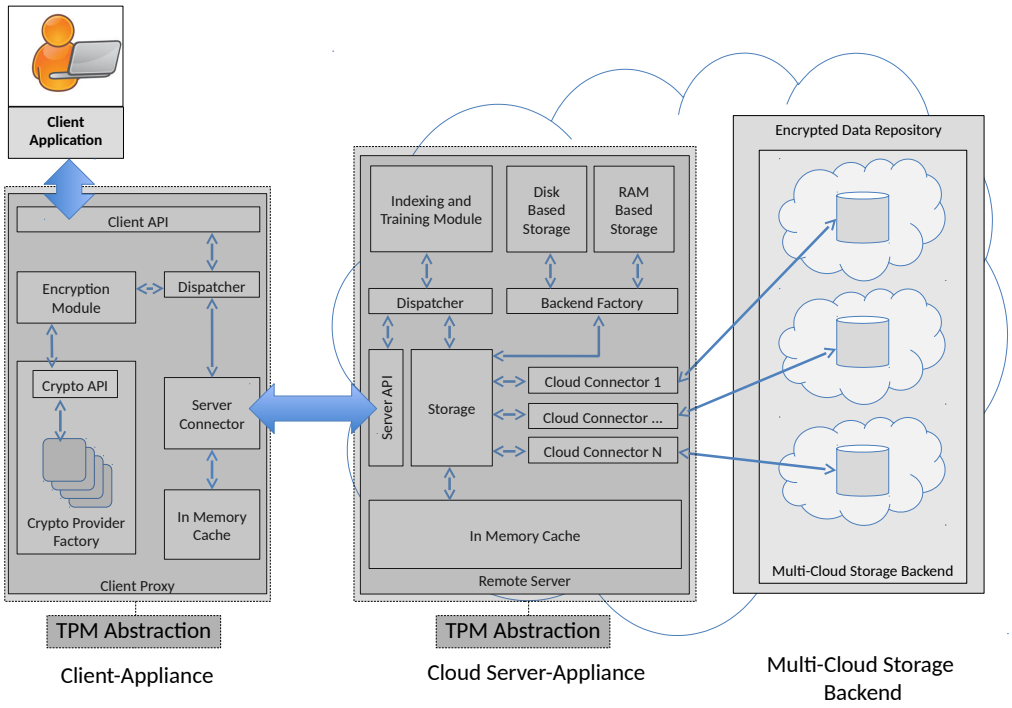


Figure 3.2: System Model in Detail

3.3.1 Client Proxy

The client proxy connects directly with the client applications that want to make use of our system. When uploading or searching data the client proxy will handle the feature extraction from text and images, the encryption of both features and data and sending the encrypted data to the middleware server. The feature extraction is a necessary step so that the middleware server can index and search on the encrypted data. When downloading the client proxy will handle the retrieval and decryption of the data from the middleware server. There is no integrity check when retrieving data as that was already made in the middleware server and will be explained in more detail in the next Section.

The client proxy supports unstructured image, text and mime documents, providing specific methods for each type. We define unstructured image and text as an image, for example a jpeg file, and a text document, for example a text file without specific structure, that are not organized in a structured way. Notwithstanding the unstructured characteristic of these files we must consider that there contents must be related in search operations. For mime documents the client proxy will parse the file and extract all text and compatible images found to index. The document itself, including all the data that couldn't be indexed, is sent to the server along with the features of the indexable data.

We refer to the association between the unstructured image and text as document, since the client proxy is the only entity that distinguishes between unstructured and structured data. For both the middleware server and the storage backend all data received to index and store represent an opaque document with a set number of images and keywords. When

an user retrieves a document that is made of unstructured data both the image and text will be retrieved. If the user retrieves a mime document the original uploaded document is retrieved.

Client API

We provide an API that client applications can use to use our system functionality. Here we will list and explain all the methods provided.

1. addUnstructredDoc

   This method receives an image and a text along with a name and sends them to the middleware server to be stored as a document. The client proxy will extract the features of the text and the image, encrypt all the data, including the features, and send it to the middleware server.

2. getUnstructuredDoc

   This method receives a name and retrieves the document associated with that name. If the client cache is being used and the document is there then it's retrieved from the cache and no communication with the middleware server is made. Otherwise the document is requested to the middleware server. Upon receiving the document from the middleware the client proxy will decrypt it and store it in it's own cache before splitting it in it's image and text components and returning it to the client application.

3. searchUnstructuredDocument

   This method receives an image and a text and sends a search request to the middleware server. The search request is composed of the encrypted features that were extracted from the image and the text. It receives a variable sized list with potential matches. The number of results can be indicated by the client application or a default can be used. We have chosen twenty matches as the default.

   Unlike the addUnstructuredDoc method, the data itself is neither encrypted nor sent as the only requirement for the search are the features. This method will return a ranked list of documents names, not the document themselves. The client application can choose to retrieve each document individually or based on other factors with the getUnstructuredDoc.

4. addMime

   This method is similar to the addUnstructuredDoc presented above, however it receives a mime document instead of an image and text. The client proxy parses the document and extracts the features of all text and compatible images. It then encrypts all features and the document and sends it to the middleware server.

5. getMime

   Like its getUnstructuredDoc counterpart, this method receives a name and retrieves the document associated with that name. Its processing is in everything similar to

getUnstructuredDoc, however the document is returned as it was sent. There is no split between images and text before returning the document to the client application.

6. searchMime

   This method is in everything similar to searchUnstructuredDoc. It will ignore any data presented in the mime that is not an indexable image or text while extracting the features of indexable data. Only those features will be sent to the middleware server. It will return a ranked list of document names that potentially match the features of the provided data.

7. index

   This method will start a training and indexing operation on the middleware server. For better search results this method should be called after a reasonable big quantity of data was uploaded as the training operation can be expensive. During the training and indexing phase it is possible to still use the middleware server as usual although indexing operations will be ignored while either a training or indexing phase is in progress. Search requests might also be slower as access to the indexes is restricted while an indexing phase is being done. The training phase, which is the most expensive of the two phases, doesn't restrict index access and search requests can be processed at normal speed. Since the training phase is considerably longer than the indexing phase, it is a relatively small period in which search requests are slower.

8. printServerStatistics

   This will return to the client application some measurements about the time operations in the server take to conclude. The server measures both processing and networking time. On the processing times it provides the train, index, and search times. Associated with the train and index time there is also the network feature and network index time. The network feature refers to how long the server took to store the features it has in memory to the storage, while the network index time refers to how long it took to store the indexes.

   Both of these times include some processing and networking components as storing data involves fragmenting and signing it before the upload to the storage cloud actually happens. For this reason two more times are measured during each upload and download operation: network upload and network download time. These refer to the time it took to upload or download data to the storage clouds, respectively. These two times are also used to measure the time user uploads and downloads took.

9. clearServerStatistics

   This method resets all the server timers to zero.

10. getCacheLimit

Returns the max size that all the documents on the client cache can have. The actual size of the cache will be bigger than this value because of the data structures required to hold and maintain the cache.

11. setCacheLimit

    The client application can set a specific size for the cache during runtime to better adapt to the client device. If this value is negative or zero the use of cache will be completely disabled until a new call to this method with a positive number is made and all documents currently in the cache will be removed. If the new size is smaller than the current size then older documents are removed until the size of all remaining documents is smaller than the new defined max size.

12. getCacheTTL

    Returns the max time that documents will remain in the cache in seconds. After this time is up the document must be retrieved from the server.

13. setCacheTTL

    Sets a new max time for documents to remain in the cache. Unlike the setCacheLimit method, when changing the max time to a lower value than the one currently active doesn't automatically remove all documents that have expired unless the new value is zero or negative, in which case the use of cache is disabled and all documents are removed from the cache.

14. clearCache

    This method removes all the cache contents but without disabling the use of cache or changing any of its parameters.

Encryption Module

The encryption module is responsible for all the cryptographic operations. In practice it acts almost as a wrapper for the cryptographic provider developed, managing and generating the cryptographic keys used. It is also where all the cryptographic settings such as encryption mode or padding are defined.

Cryptographic Provider

In addition to our system we also developed a cryptographic provider capable of being integrated with the Java Cryptography Architecture. We implemented all the required methods and followed the standard defined for providers, allowing out provider to be integrated transparently. Requesting a cipher instance for CBIR encryption is done just like requesting a cipher for any other cryptographic algorithm, for example AES.

Despise following all the standards, some things needed to be different due to the nature of CBIR itself. While CBIR encrypts the features in a way that makes them searchable, the original data needs to be encrypted with a standard encryption algorithm. We allow the application making use of our provider to choose the algorithm, mode and padding to

be used with the use of algorithm parameters passed when initializing the cipher instance. Creating these parameters requires only the application to construct an AlgorithmParameterSpec passing as argument to the constructor the transformation just as if requesting that transformation from the Cipher class. It is important to note here that there are two CBIR algorithms, one for images and one for text. The difference between the two is explained in great detail in [53], however this lead to two implementations. When instantiating specific classes of the CBIR algorithm it's important to use CBIRD for images, and CBIRS for text. For example, to request a cipher instance to encrypt an image we would use CBIR**D**WithSymmetricCipher while to request a cipher instance to encrypt a text document we would use CBIR**S**WithSymmetricCipher. For the application they are otherwise equal.

Additionally, since we are following all the standards from JCA, the return of the two cipher had to be merged. We don't expect a programmer to know how the resulting byte array is organized and as such we provide an utility class that receives the array returned by the cipher and interprets the result, providing methods to access each of the cipher texts. Although we use a single key to encrypt all features and documents, an scheme like the one presented in [53] in which there is a repository key used to encrypt the features and each document is encrypted with its own symmetric encryption key is also possible. The encryption keys used on the CBIR algorithm with a symmetric cipher are composed of two elements, the CBIR key and the symmetric cipher key. Each of these keys can be generated independently, and then both can be used together to create the key required for the cipher instance.

For cases in which the encryption of the data is not required, for example for searches, we also provide an instance to use only CBIR without any other encryption. This interface however uses the MAC class instead of the Cipher. This is because the MAC interface was better adapted for the CBIR requirements.

In Memory Cache

The cache stores recently retrieved and decrypted files from the server. This allows the client device to reduce processing work when requesting the same documents in a short time. Both the total size of the documents and how long they are considered fresh can be configured and changed at runtime to better adapt to the needs of the client application and device.

When removing documents from the cache we remove the oldest first. An auxiliary control data structure keeps an ordered list of references to the items to remove next reducing the processing required when removing items. By removing the oldest items we keep the complexity of the cache low as the same time as taking advantage of the TTL defined. The TTL is not refreshed when an item in the cache is accessed which makes the older items have a bigger probability of being removed due to being expired. Using an algorithm like LRU could potentially lead to the unnecessary removal of documents if for example the cache needs to make room for another document and the least recently used document is not the oldest. That document would be removed, the new document inserted, and then the oldest document expires leading to its removal.

Server Connector

The server connector sends the requests to the server after the data is processed and receives the result to queries and retrieval requests. Since all the data is already encrypted we use regular sockets with TCP to communicate with the server. The server connector is also responsible for compressing and decompressing the data sent and received over the network.

3.3.2 Middleware Server

The middleware server is where indexing data is stored and all searching related operations are made. It has the biggest computational requirements of the three main blocks as the training operation can be very expensive and the features data might take a lot of RAM on top of being composed of several other complex components.

We use two storage types, disk and RAM, and for each type a specific component is used. For disk based storage we use Depsky [12] which provides fragmentation, integrity checking, encryption and secret sharing. Since we use our own encryption which is done in the client we only use the fragmentation and integrity check implemented by Depsky. We don't use secret sharing as it's better used to share the encryption keys, however to do so the server would need to have access to the key used, which would in turn give it access to all the data if it was compromised. In this mode only our middleware server, the DepSky client and possibly the cache will be running on the middleware server machine.

For RAM based storage we use RAMCloud [13]. RAMCloud doesn't provide fragmentation or integrity checking so we did our own implementation of it by using OpenSSL [54] for public key cryptography and Jerasure [55] for fragmentation. In this mode the middleware server as well as several RAMCloud clients, the ZooKeeper server and cache will be running on the machine.

Middleware Server API

The server provides an API through TCP sockets to the client. To select an operation a client sends one byte at the start of the connection and then follows the operation semantic. There are six operations available to the client: addDoc, search, getDoc, index, printStatistics, clearStatistics. The semantics of operations were made with the intent of reducing round-trip times while maintaining guarantees of correctness. All requests receive a response with a status code indicating if the request was accepted, succeeded or failed. A request can fail if the server is in shutdown mode, in which case it is waiting for previous requests to complete, but its not accepting any new requests.

The client will start sending data to the server before receiving a response, which can lead to some unneeded network traffic, however this allows requests to be handled in one round trip time. Waiting for a response from the server before sending the data would increase the time it would take for all requests to be made. Since the server is expected to accept all requests, then the possibility of significant extra network traffic is small. In cases

where the request has extra data for the response, such as getDoc, search and printStatistics, the status code is sent together with the response.

1. addDoc

   This operation is invoked with the byte 0x61 ('a'). The client sends the document and encrypted features and waits for a confirmation of the server that it was received and accepted. The client doesn't wait for a confirmation of the write finishing as under our adversary model it will always succeed. It would also slowdown the upload as the client would need to wait for the storage and indexing to finish.

   The server will then send the encrypted document to the storage module for processing and storage, and store the features in memory for the index operation. It should be noted that the features will always be stored in memory as they might be used in a future training phase even if a previous training phase was already done.

2. search

   This operation is invoked with the byte 0x73 ('s'). The client sends the encrypted features and the server searches for documents that match those features. If the index is built then it is used, otherwise a linear search on the features in memory is made. This linear search is considerably slower and should be avoided as it iterates all documents features stored in memory. However, the precision of the index search is related to how many features were in memory at the time of the training phase, which can be slow.

   The best precision is achieved by uploading a large amount of documents and doing a training phase afterwards before invoking many searches. It's worth noting that the precision might fall slightly when uploading documents after the training phase has been done. In that case, after a large amount of documents have been uploaded, another training phase should be done to keep precision high.

3. getDoc

   This operation is invoked with the byte 0x67 ('g') and its followed by a flag byte. If the flag byte is 0x66 ('f') the request will ignore the server cache. The client sends the name of the document to retrieve and the server retrieves it through the storage module. Reconstruction of the document and integrity check are all done on the server. The client receives the encrypted document ready to be decrypted. If no document is found with that name a not found error code is sent. Integrity check is done on the individual fragments retrieved from each cloud and not on the reconstructed document. This allows the server to start the reconstruction of the document before having all the fragments ready as long as it has the minimum amount necessary for reconstruction. The erasure code algorithm will provide the guarantee that the reconstructed document is the same as the original if none of fragments used are corrupted. Although

the request is made, the storage clouds with the highest latency are ignored, as long as the fragments on the other clouds are valid.

4. index

   This operation is invoked with the byte 0x69 ('i'). This operation can have different effects based on the status of the server. If there are features in memory but the codebook is not generated yet it will start a training phase, which is followed by an indexing phase, otherwise only a indexing phase is done. At the end of the training phase the current in memory features are stored and after the index phase the index is stored. A training phase can be forced even a previous training phase was already done to restore precision after a big quantity of documents have been uploaded.

   This is the only operation that stores the features in the storage backend. This is because storing the features in each upload would be too expensive as there isn't an append function in the storage backend that allows the new features to be written without retrieving the features already stored first.

5. printStatistics

   This operation is invoked with the byte 0x70 ('p'). It will send all the time measured by the server to the client up to that point. It doesn't affect the timers as this allows more flexibility in time measurement. A client can issue this command in the middle of a sequence of requests to measure the time up to that point and then a final one to measure the full sequence. It will send processing times as well as network times for all operations, allowing the client to see where is the server spending more time. A more detailed description of the times was presented on the client API Section for the respective method.

6. clearStatistics

   This operation is invoked with the byte 0x63 ('c'). It will reset all the measurements to zero allowing the client to easily discard the times measured on the previous operations.

Indexing and Training Module

To search on encrypted data we keep several indexes in memory along with the encrypted features of all documents stored in our system and a codebook. Features are encrypted with CBIR [53] on the client side while data is encrypted using AES. All operations are done on encrypted data as the server never gets access to the encryption keys used. This also guarantees the privacy of the user data as the only component to see the plain text data is the client proxy.

Keeping the features in memory can represent a significant overhead in memory usage. To deal with this it we only keep in memory the features of data uploaded after the last indexing phase. Features from data from before the last indexing phase are already indexed

and don't need to be in memory since we won't do a linear search over them. Storing them on each upload would add an extra overhead as the old file would need to be retrieved, the new data added and then stored. Over time this would lead to unacceptable delays for each upload. Instead we keep old features on storage, retrieve and update that file only on the indexing phase which is much less common than uploading and leads to minimal disruption for the user.

To index the features we use the codebook created during the training phase. This codebook is also used for the search and so must be kept in memory. Once a training phase is completed we lock access to the indexes, update to the new codebook and index all the features. During this period the user applications can't do searches, although other operations still work. For optimization we don't move the features from one structure to another, but instead we move the structure itself and create a new one to store the new features. This allows this move to take constant time instead of being linear with the amount of features in memory and increasing the time the indexes are locked. Once the indexing is done the old structure is discarded.

To prevent serious overheads on memory usage we keep track of how many features and how much DRAM they are taking. Once it hits a pre-configured threshold it triggers an automatic training and indexing phase as a way to remove the features from memory. This however is not intended to be a replacement for user requested indexing as several other factors might make a new training and indexing recommended. For example, if many images were uploaded the precision might fall enough for users to start noticing. However this isn't a guarantee that the features of those images will trigger the high memory usage threshold. This threshold should be high as to not cause needless training and indexing phases which can be disruptive to the user usage of our system.

Storage

The storage module is responsible for managing all the reads and writes done to the storage backend as well as access to the cache. It keeps track of files currently being written and manages concurrent accesses to the same file by different threads. We allow one writer and multiple readers. To prevent starvation of either readers or writers we use a model in which writer threads lock the file preventing any other thread from reading or writing. Readers that try to access the file without a writer holding the lock are placed on a readers group. When a writer arrives it locks the file and any future readers will be placed on a waiting readers group. The writer then waits for the readers group to be empty and starts the write. Once the write is done it will see if there is any reader on the waiting readers group, if there is it will signal them and release the lock. Once the lock is released any new readers will be placed directly into the readers group. Readers on the waiting group will be moved to the readers group, and when the last reader is moved it will signal the next writer. The writer will then lock the file and wait for the current readers to finish. If when the write is done the waiting readers group is empty the current writer signals directly the next writer before releasing the lock. This ensures that both readers and writers can always make some progress.

To control the memory usage of writing or reading big files (on some of our tests we had a features file of 1.7GB) we define a maximum file size so we can have a bounded upper limit of how much memory our system will use for any read/write. Every write will internally split the files into equally sized segments if the file size is bigger than the maximum allowed, although the storage module offers a way to prepare a write of a big file ahead of time. In this case the size of the file must be known beforehand so it is possible to split the segments correctly. Afterwards we can pass parts of the file for the write operation in sequence, instead of sending the whole file in a single call. This has the advantage of avoiding allocating a big array in memory, using instead many small ones that don't need to exist at the same time and reducing the maximum memory taken by the write. We use normal writes for all user operations since the default maximum file size is 350MBs which should be enough for most unstructured documents of images and text or mime documents. For our system reads, namely the features, codebook, and indexes we use a prepared write. We store the information about big files in a special blobs file. This file is kept in memory and also updated on storage after every write that triggers a division and as such is not ideal for use in user reads or writes, although it can, and will, be used if a user uploads a file considered big. When reading we consult this file to know if we read the whole file at once or in parts.

Because of this the storage module offers two kinds of writes/reads. A partial read/write in some ways similar to a stream and a single read/write operation that operates on the whole document. The single write operation will split the file internally, the partial write will keep the data in memory until it reaches a limit. When that limit is reached the data in memory is sent to be stored. As a way to prevent inference we divide the file into equally sized smaller files. This is possible even with partial writes as we know the size of the file ahead of time. The partial write main use is to prevent situations in which we have to prepare a large amount of data to be stored, like the features file. Instead of preparing and allocating all the data in a single large array, we send it piece by piece as soon as each piece is ready using less memory overall. The partial read retrieves the file by reading each of the smaller files as needed. Both partial writes and reads are prepared to deal with files smaller than the max file size. In this case no file division is done.

Backend Factory

The backend factory is where we make the division between RAM storage and disk storage. The two available backends are RAMCloud and Depsky. The backend factor is responsible for fragmenting the file and signing each fragment before the upload. For this purpose we use Reed-Solomon codes for fragmentation allowing for the reduction of file sizes on each cloud by half. This results in using twice the storage across all clouds, instead of four times as we would by replicating the whole contents of the file. The use of Reed-Solomon codes allow the recovery of the original file with only two fragments although we still need a quorum to read the metadata and retrieve the information required for the decoding. We sign those fragments and check their integrity before starting the decoding process. Any fragment that fails the integrity check is discarded and we treat it as if that cloud doesn't have the fragment.

The requests for the clouds are done in parallel and for optimization we assume the operation succeeded after three of the four requests finish. This follows our adversary model in which one of the storage clouds can be unavailable or compromised in some way. While we don't wait for the fourth request to finish, we don't cancel it. The request completes on the background and silently terminates. This allows us to ignore the storage cloud with highest latency when uploading and downloading if the remaining clouds are working correctly and start to freeing some of the resources used by those operations sooner.

In Memory Cache

We use Memcached to reduce the number of requests to the storage backend and the processing done to reconstruct the files. This allows much faster responses to the client. The cache can be disabled on a configuration file or on a user request basis, which will force that request to ignore the cache and go the backend. This request won't affect the others done and although it ignores the cache when reading the file, it will update the cache contents after the file is retrieved.

Cloud Storage Backend Connector

The backend connector is responsible for dealing with requests to each individual storage cloud. Each cloud service provider will have a specific one when using disk storage as they will use the API provided. To add a new service provider for a disk storage cloud the connector would have to be developed. For RamCloud there is only one connector regardless of service providers as it will use the RamCloud API.

3.3.3 Cloud Storage Backend

The cloud storage backend is a cloud-of-clouds, allowing for dependability and availability even when some of the individuals clouds are unavailable. For disk storage backends we use regular storage clouds like Amazon S3 [56] or Google Cloud Storage [57]. For RAM storage we use servers like Amazon EC2 [58]. Disk storage backend is possibly cheaper and with more capacity. RAM storage offers more speed but at a more expensive price.

3.4 System Operation

In this Section we will describe the steps taken when each operation is performed. We will divide them into four main groups: add, get, search, index. On each group we will explain the algorithm on the client side, and then on the server side.

3.4.1 Upload

The algorithm to add an unstructured document on the client side is described in algorithm 1. The algorithm for adding a mime document is described in algorithm 2. The difference between the two is that when adding a mime document the dispatcher parses the document to select all indexable parts of it. Then it forwards those parts to the encryption module to extract and encrypt the features of each part. The document is then encrypted. To add an

unstructured document the dispatcher forwards the input directly to the encryption module. The encryption module will then extract and encrypt the features as well as encrypting each part individually. Algorithm 3 describes the operations executed in the server once it receives an upload request. In the server there is no distinction between mime and unstructured documents. The division is simply between features and user data which the dispatcher forwards to the index and training module and to the storage module respectively.

```
   Input: An img and txt
1  begin Dispatcher
2      begin Encryption Module
3          begin Criptographic Provider
4              Features← ExtractFeatures(img);
5              EncFeats← EncryptFeatures(Features);
6              EncData← EncryptData(img);
7              return CipherText← JoinData(EncFeats,
                   EncData);
8          ImgsCipherTexts.Append(CipherText);
9          begin Cryptographic Provider
10             Features← ExtractFeatures(txt);
11             EncFeats← EncryptFeatures(Features);
12             EncData← EncryptData(txt);
13             return CipherText← JoinData(EncFeats,
                   EncData);
14         TxtsCipherTexts.Append(CipherText);
15     begin Server Connector
16         Data← Compress(ImgsCipherTexts,
               TxtsCipherTexts);
17         Send('a', ImgsCipherTexts, TxtsCipherTexts);
```

Algorithm 1: Algorithm for uploading unstructured documents on the client side

```
   Input: A mime document m
1  begin Dispatcher
2      imgs,txts ← ParseMime(m);
3      begin Encryption Module
4          foreach img do
5              begin Cryptographic Provider
6                  Features← ExtractFeatures(img);
7                  EncFeats← EncryptFeatures(Features);
8              ImgsCipherTexts← Append(EncFeats);
9          foreach txt do
10             begin Cryptographic Provider
11                 Features← ExtractFeatures(txt);
12                 EncFeats← EncryptFeatures(Features);
13             TxtsCipherTexts← Append(EncFeats);
14         begin Cryptographic Provider
15             CipherText← EncryptData(m);
16     begin Server Connector
17         Data← Compress(ImgsCipherTexts, TxtsCipherTexts,
               CipherText);
18         Send('a', Data);
```

Algorithm 2: Algorithm for uploading mime documents on the client side

```
1  begin Dispatcher
2      DecompressedData← DecompressData;
3      EncryptedData← Parse(DecompressedData);
4      begin Storage
5          EraseCacheEntry(name);
6          Fragments← Fragment(EncryptedData);
7          foreach fragment do
8              Metadata← Hash(fragment);
9          Sign(Metadata);
10         Store(Fragments, Metadata);
11     EncryptedFeatures← Parse(DecompressedData);
12     begin Index and Training module
13         foreach feature do
14             InMemoryFeatures.Add(feature);
```

Algorithm 3: Algorithm for uploading documents on the server side

3.4.2 Get

The algorithm to retrieve a document from the storage is presented in algorithms 4 and 5 for unstructured documents and mime documents respectively. Like the add algorithms the difference is between the parsing made of the data. To retrieve an unstructured document the dispatcher parses the encrypted document and the encryption module decrypts the image and text individually. This is possible because together with the encrypted data some metadata was also stored. This metadata is limited to the size of the cipher texts,

which is public information for anyone who would access the cipher text. To retrieve a mime document the dispatcher forwards the encrypted document directly to the encryption module. The algorithm from the server is described in algorithm 6. From the server side there is no difference between the two types of documents. The server will retrieve at least three of the fragments, verify their integrity and reconstruct the original encrypted document.

```
   Input: name
1  begin Dispatcher
2      if name ∈ cache then
3          Document← RetrieveFromCache(name);
4      else
5          begin Server Connector
6              EncryptedDocument← Retrieve(name);
7          begin Encryption Module
8              img← Decrypt(EncryptedImg);
9              Document.Append(img);
10             txt← Decrypt(EncryptedTxt);
11             Document.Append(txt);
12         Cache.Add(name, Document);
```

Algorithm 4: Algorithm to download unstructured document on the client side

```
   Input: name
1  begin Dispatcher
2      if name ∈ cache then
3          Document← RetrieveFromCache(name);
4      else
5          begin Server Connector
6              EncryptedDocument← Retrieve(name);
7          begin Encryption Module
8              Document← Decrypt(EncryptedDocument);
9          Cache.Add(name, Document);
```

Algorithm 5: Algorithm to download mime documents on the client side

```
1  begin Dispatcher
2      if name ∈ cache then
3          EncryptedDocument← RetrieveFromCache(name);
4      else
5          begin Storage
6              Fragments← Retrieve(name);
7              foreach fragment do
8                  VerifySignature(Metadata);
9                  if Metadata is valid then
10                     Verify(fragment);
11             if valid fragments > 3 then
12                 EncryptedDocument← Reconstruct(Fragments);
13                 Cache.Add(name, EncryptedDocument);
14     Send(EncryptedDocument);
```

Algorithm 6: Algorithm to download documents on the server side

3.4.3 Search

For the search the algorithm on the client side is identical to the add algorithms 1 and 2, except that the document itself is not encrypted, only the features are encrypted and sent to the server. The algorithm for the server side is described in algorithm 7. The server will check each of the features received and assign scores to matching features that are indexed. It will then merge and sort the different results from the image and text features and send them to the client. Since the list is already sorted the client only have to parse the data received from the server.

3.4.4 Index

The index operation is executed completely on the middleware server. The client only sends the operation request which consists of two bytes, one for the operation selection and a

```
1  begin Dispatcher
2      begin Index and Training Module
3          foreach image feature do
4              ImgFeatures← AssignScore(matching indexed feature);
5          ImgFeatures← Sort(ImgFeatures);
6          foreach text feature do
7              TxtFeatures← AssignScore(matching indexed feature);
8          TxtFeatures← Sort(TxtFeatures);
9          Results← Merge(ImgFeatures, TxtFeatures);
```

Algorithm 7: Algorithm for searching on the server side

second one to force a training phase if required. The algorithm is described in algorithm 8. The server will start by switching the containers where the features are stored in memory so that uploads can still be processed during this operation. It will then proccess all image features, both in memory and in storage to create a new codebook. After the codebook is created all the features are indexed using the new codebook. The text features will be indexed and stored directly without a training phase. After both image and text features are indexed the indexes are stored in the multi-cloud storage backend.

```
1  begin Index and Training Module
2      TmpImgFeatures← InMemoryImgFeatures;
3      TmpTxtFeatures← InMemoryTxtFeatures;
4      InMemoryFeatures← new empty feature container;
5      if image features in storage then
6          foreach image feature in storage do
7              if RNG <= 10% then
8                  Trainer.Add(image feature);
9              if image features in storage fit in memory then
10                 TmpImgFeatures.Add(image feature);
11     foreach image feature do
12         if RNG <= 10% then
13             Trainer.Add(image feature);
14     codebook← Trainer.Train();
15     Store(codebook);
16     if image features in storage and did not fit in memory then
17         foreach image feature in storage do
18             codebook.Index(image feature);
19     foreach image feature in TmpImgFeatures do
20         codebook.Index(image feature);
21         Store(image feature);
22     foreach text feature in TmpTxtFeatures do
23         Index(text feature);
24         Store(text feature);
25     Store(ImgIndex);
26     Store(TxtIndex);
```

Algorithm 8: Algorithm for training and indexing

3.5 Architectural Options for Deployment

Because of our division of the system in main blocks and the way they interact with each other there some possibilities to the its deployment. The client proxy will always be deployed together with the client application, however the middleware server can be deployed on local server close to the client proxies, or even in the same machine as a client proxy, or it can be deployed in the cloud. In case of RAM based storage the backend can also be deployed on a company servers without any changes. For disk based storage that would require

the implementation of the storage server itself and the connector for our server to use the storage server API. On this Section we focus on the options available for the middleware server, since it can't be replicated and can be the bottleneck for the system performance.

3.5.1 Local Behaviour

When deployed locally with the client application the middleware server will offer the best possible latency for searches and gets from the client that hit the cache. The clients will essentially have two caches, if both are enabled. However uploads, and operations from clients that are not local, might suffer from it and become slower. This local server will also have to have the capability to support the middleware server load which can put a considerable strain on its resources.

3.5.2 Cloud Behavior

By deploying the middleware server on a cloud the latency to the clients will increase, however it is possible that it can offer a better performance than deploying it locally. This is because although the latency to the client increased, the latency to the storage backend will probably decrease. This makes the time the server spends executing the upload less, reducing the time resources are being used on the same request, which in turn can be used to do other client requests. However the longer a client takes to upload the data, the longer the server has resources allocated for that request without doing any work, so there must be a balanced approach when looking at where the middleware server is deployed.

3.5.3 Multi-Cloud Behavior

Although we make use of several storage clouds to guarantee availability of the data, we don't replicate the middleware server. There are several problems with the replication of the middleware server addressed in detail in the final chapters as future work. Another possibility is to keep several middleware servers in stand-by for the case of a crash and keep only one active. Although that presents less problems than replication it is not a direct deployment. Because features are only stored during the training and indexing phase, if the active server crashed some features could be lost, which would affect the next training phases and by consequence the precision of searches for those images until they were re-uploaded. We also don't provide a mechanism to detect a server crash and let a second server take over, communicating this change to the clients. This would provide an extra guarantee of availability, although at the possible cost of search precision.

3.6 Discussion of Architectural Variants

All deployments presented in the previous Section have advantages and disadvantages based on the situation that they are used. To serve a limited number of clients all in close

proximity to each other, for example the employees computers of a company in a single building it might be beneficial to deploy the middleware server on a local server dedicated to it. The server cache would reduce the time for retrieving a document across the clients, while the client cache would further reduce it for each client. Employees could even access the middleware server from their smartphones or tablets if they are connected to the company network.

However this solution would not be practical if the employees were to have access to the middleware server from their home as it would mean either open the middleware server to outside connections or the employees would have to be able to connect to the company network from outside, so they could then connect to the middleware server. In this situation a cloud deployment would provide more advantages than a local deployment. It could be accessible from outside the company network without exposing it and it would benefit from the scalability of the cloud servers allowing the company to better adapt the server specifications to their needs.

The third variant with multiple instances of the middleware server would improve the reliability of the system, however it would increase the latency of the operations due to the synchronization, either from the multiple instances or to allow recovery without precision loss.

3.7 Summary and Concluding Remarks on the System Model Design

Our system is designed in three main blocks, each with their own adversary model. This allows for several deployment schemes, each with their own benefits and trade-offs. The different adversary models allows our system to deal with different problems in different blocks. This makes it easier to move the more expensive operations to a single block which is deployed in a more powerful device, while the operations responsible for data privacy can remain on the client device, which might be a smaller device, like a smartphone.

The three main blocks of our system are the client proxy, the middleware server and the multi-cloud storage backend. The client proxy is deployed in the client device and communicates directly with the applications. It offers methods to encrypt, upload, search and download unstructured images and text documents, or mime documents. The middleware server is deployed in a local server or cloud and is responsible for the indexing and search operations. It will also fragment and replicate the data to the multi-cloud storage backend when uploading and reconstruct the file from the fragments when downloading. The multi-cloud storage backend consists of four clouds which hold the file fragments. Only three clouds are needed to reconstruct a file and we assume that one of the four clouds might be compromised in anyway.

Files are encrypted on the client proxy, and none of the other blocks has any access to the encryption keys used. Although communication from client proxy to the middleware server

is made unencrypted using sockets all the user data and indexable features is transfered encrypted. The adversary model for the middleware server and the multi-cloud storage backend allows an attacker to have access and inspect files and memory contents.

CHAPTER 4

IMPLEMENTATION

In this chapter we present the implementation details of our system [1]. We will present about the technologies used in each of the three main blocks as well as how each module of each block is implemented and connects to the other modules. We will also describe the implementation and test environments used. Additionally, we will also explain our criptographic provider implementation and how it integrates with the JCA framework and standards and give some examples of it's use.

4.1 Implementation Environments

Our system is developed in Java and C++. We used Java version 1.8.0_91 and g++ 6.1.1 for compilation. Our system also requires several external libraries to work. We used RamCloud, commit from 23th May 2016 [59] and Depsky, commit from 16th June 2015 [60] for the backend communication. For fragmentation with Reed-Solomon codes we used GF-Complete 1.03 and Jerasure 2.0 [55]. Jerasure is only used for RamCloud as Depsky already provided it's own implementation of erasure codes. For feature extraction of images we used OpenCV 2.4.10 for the client and OpenCV 3.0 for the server. For integrity check with RamCloud we used OpenSSL 1.0.2h. We integrated Depsky with our C++ code using JNI. The client proxy is completely done in Java, while the middleware server is done in C++ with the exception of the Depsky code. We also leverage Trousers 0.3.13 [61] with the TPM emulated by TPM-Emulator 0.7 [62] for remote attestation of the middleware server code.

[1] The implementation is available at https://github.com/khasm/seasky

4.2 Technology

In this Section we will explain the implementation details of each component and how they interact with other modules in the same block. We will start by the client proxy and then the middleware server. This section won't cover the multi-cloud storage backend as that consists of either storage clouds provided by cloud services or the RamCloud servers which were not implemented by us. Although we tried to not alter either the Depsky or RamCloud code some changes were required. For Depsky the changes were limited to adding methods to the DepSky API so we could access the time measurement for uploads and downloads, as well as reset those times and the selection of the region in which buckets were created in the Amazon Driver. For RamCloud we increased the timeout before a RPC was aborted and retried as the higher latencies made several requests abort and retry. Although the request ultimately succeeded as the storage server was working and had received the request the first time this lead to less false positives of server crashes and network traffic.

4.2.1 Client Proxy

The client proxy consists of four classes, each representing a module, not counting the cryptographic provider. The dispatcher implements the API available to the client applications and forwards data to the other modules as needed. The client application creates an instance of the dispatcher which will create instances of the other classes for it's use. The dispatcher makes use of javax.mail to parse the mime documents and get images and text. Currently only images with content type image/jpeg are extracted, all other images are ignored. Text only needs it's content type to start with text to be extracted.

Client API

The client API is implemented in a single interface designed to be simple and easy to use. It receives byte arrays as arguments, leaving the method of obtaining a document contents to the client application. After the client application has the contents of the document all processing is done by the client proxy without the need for the client application to do anything else. When retrieving documents the client proxy returns them as byte arrays.

Encryption Module The encryption module is responsible for managing the cryptographic functions. It will read the keys from disk, or if they aren't found generate new ones and store them. It's other functions will act as wrappers for the cryptographic provider. Since the encryption module isn't intended to be used by the client application directly we don't use the utility class on the cryptographic provider to wrap the cipher text resulting from the CBIR and AES cipher. Instead we return those cipher texts directly to the dispatcher and parse them on the server connector. To improve performance the encryption module provides functions to encrypt the documents received with both CBIR and AES or only CBIR. This allows searches to be performed without using AES since the AES cipher text is not used on that operation and also allows the encryption module to encrypt the features of images and text of mime documents while not having an overhead of encrypting

the image or text with AES twice, or splitting the mime document into several parts which would then have to be tracked to reconstruct the original document when retrieving it. For unstructured documents we use CBIR with AES, which performs the feature extraction and encryption as well as AES encryption in a single method call.

In Memory Cache

The in memory cache is implemented as a map with the document names as keys and cache entries as values. Cache entries consist of the document contents and a time value indicating when that document was retrieved. That time value is used to know when a document has become too old and must be discarded. When the cache is full we discard the older documents first as they are more likely to be removed sooner than new documents. To avoid iterating over the map every time we need to add a document to the cache and another document must be removed we also keep a queue with the name of the documents. When a document is put in the cache it's name it's also put at the end of the queue. When removing files we remove the files at the head of the queue until the new document can be placed in the map without violating the maximum memory permitted for documents. Documents that are rarely used will eventually be removed as the older documents are renewed and placed at the end of the queue. Documents that are used many times will be removed when their time expires or the cache needs to free some memory for a new document. The second case should not happen often if the cache size is set up properly since all documents will at some point be refreshed and be put at the end of the queue as long as they are being requested. Client applications can set up the cache size and expire time through the API to better adapt the cache to their needs.

Middleware Server Connector

The middleware server connector contains all the protocols to communicate with the server API. It uses sockets and byte buffers to parse and format the requests. It also compresses and decompresses the requests and answers.

4.2.2 Middleware Server

The middleware server is composed of six modules: the dispatcher, the indexing and training module, the storage module, the backend factory, the in memory cache and the cloud connectors. Similarly to the client proxy the dispatcher receives and redirects the data as needed. Cryptographic functions in the middleware server are reduced to hash and signatures. All the user data processed is encrypted on the client proxy and there isn't any situation in which the client proxy will send the encryption keys to the server. The server has it's own public key pair to sign and verify the integrity which are stored on the middleware server disk.

Middleware Server API

When a request is received in the middleware server API, a thread is assigned to handle that request. If no threads are free then the request has to wait until a previous request completes. The number of threads depend on the number of available cores available to the

middleware server. Those threads are created when the middleware server starts and are only terminated when the middleware server stops. When a request is received the first byte the thread checks the first byte and calls the respective function. If the byte is not recognized as a valid request the connection is closed and the request ignored.

Indexing and Training Module

The index and training module handles everything related to indexing and searching. Features are kept in memory up to a maximum, if a new upload would take the memory taken by the features above the maximum a self request to index is made to the server API, causing another thread to start the index processing. After the index is in progress and the feature containers are switched by new ones the upload proceeds as normal. This causes the new document to not be indexed by the automatic indexing, but guarantees a maximum memory consumption regardless of the size of the features being uploaded.

When indexing features are handled using a process similar to a stream. It first tries to read the features in the storage backend to create the codebook. Those features will be cached in a secondary feature container if their total size is small enough to fit in the maximum allowed memory for features. This parameter is different than the maximum memory allowed for features in memory outside an indexing phase and can be configured separately. If the features are too big then they are discarded and a second read from the backend will be made for the indexing phase. This is required because the new indexing will be done with a different codebook. If the features are kept in memory then the second read isn't done. Features will then be written as they are indexed. While the index and codebook are read automatically when the server starts, the feature files are only written or read specifically during the index and training phases.

Storage Module

The storage module makes the connection between all the other modules and the storage backend, as well as controlling all the concurrent access to files. It keeps an list of open files and the threads that are using them. These file entries have a buffer for each reader thread and a single buffer for the writer thread. The current writer thread is identified by a thread id which allows all threads to make use any kind of reads or writes without worrying about concurrency. The storage module will check if the thread attempting to write is the current writer thread, or if there isn't any other thread writing and allow or delay the write accordingly. The buffer sizes will be determined by the file size, but each buffer will have at maximum the size of the maximum file size allowed. Files bigger than that are split in several parts and only one part is kept at the same time for each thread. Because bigger files can take some time to write readers are still allowed to read while a thread is writing under some circumstances. Reads are only stopped when a write enters the commit phase. For small files that aren't split in parts, the commit phase begins as soon as the write begins. For bigger files a list of the previous parts of the file is retrieved, then new parts are named, making sure to avoid any name conflicts with the previous parts. Those parts are then written to the storage backend while readers are still allowed to read the older parts. Once all parts are written the commit phase begins and readers are prevented from starting more

reads. During the commit phase the metadata is updated to reflect the new parts of the file and the old parts are removed from the storage backend. The storage module also manages which files are stored in the cache and when to retrieve a file from there. When reading big files only parts of the file are attempted to be placed and read from the cache.

Backend Factory

The backend factory creates an abstraction between RAM and disk based storage. In the case of disk based storage the backend factory serves only as a wrapper to call the JNI functions that connect to the JVM running Depsky. It also keeps track of the threads and environments to which each thread is attached. Since our thread pool doesn't create new threads for new requests we only keep track if a thread is already attached and to which environment. We don't detach threads at the end of a operation as the same thread will potentially make another operation and will only cease to exist when the middleware server terminates.

For the RAM based storage the backend factory implements the fragmentation of the documents as well as the hash, signing and integrity checks. Requests are handle in a similar way to Depsky. A thread is responsible for processing all operations to a storage cloud. These requests are serialized, however requests to different storage clouds are done by different threads and so are done in parallel. When a thread from the dispatcher wants to read or write a document it will put the request in the storage cloud queues and signal the responsible threads that a request is waiting. Threads managing the queue process requests until the queue is empty and them block waiting for more requests. RamCloud clients are not thread-safe so each thread must manage a client instance context to avoid synchronization with other threads.

When writing a document the metadata of the document is retrieved, if it already exists to determine the new version. This metadata is not the same as the big files metadata. Big file metadata is stored in a specific file that has information about all big files files in our system, which consists in the original file size, the size of the parts and the sequence numbers used to generate the parts names. The metadata stored with the file itself consists on the file size, the reed-solomon parameters used, version of the file, and a list of fragments that are stored in each storage cloud with their hashes. In case of big files the file size of this metadata is the size of the part of the file. The backend factory doesn't make any distinction between big or small files. This metadata is not necessarily equal across all storage clouds. The list of fragments only contain information about the fragments stored on a specific cloud. In practice that list contains only one element as we store only one fragment on each cloud.

One major difference between DepSky and RamCloud is that RamCloud has a file size limit of 1MB. This forces us to additionally perform a second split of files bigger than 1MB when using a RamCloud backend. Although this causes an additional overhead for RamCloud, it is possible to avoid the biggest overhead of this processing, which would be write each part individually causing several RPCs to write a single file. RamCloud provides a multi write rpc which writes several objects using only one RPC which we use to write the objects resulting from this second split in bulk. This allows the overhead to be reduced to

the split. For files smaller than 1MB we don't write this extra metadata.

In Memory Cache

The in memory cache is provided by Memcached. The Memcached server executes separately from our middleware server which allows it to be moved to a different machine if required although this will impact the performance of the cache. Our server expects the Memcached server be executing in the same machine unless a different IP address is written in the configuration file. We use sockets with the binary protocol to make the requests.

Cloud Storage Backend Connector

The storage backend connectors implement the required methods to communicate with the storage clouds. In the case of DepSky they use the API provided by the cloud providers and when changing from one cloud provider to another the new connector must be implemented. For RamCloud the connector is a client that communicates with the RamCloud servers. In this case the cloud provider doesn't define the API used and changing from one provider to another doesn't require a new connector.

4.2.3 Cloud Storage Backend

For disk based storage the storage backend are regular storage clouds like Amazon S3 [56], Google Cloud Storage [57] or Microsoft Azure Blob Storage [63]. They are used as provided without any alterations. For RAM based storage the storage backend are computational clouds like Amazon EC2 [58]. In this case a storage cloud is abstracted into clusters as a single computational cloud doesn't provide enough RAM to have comparable storage capacity to a regular storage cloud. For RamCloud we use a single ZooKeeper instance that executes together with the middleware server and four clusters, each with a single table where all the objects are stored. Coordinators are deployed on the storage backend together with all the storage servers for that cluster. Each RamCloud server registers itself with the ZooKeeper instance and the backend connectors retrieve the correct IP addresses from there. This allows a lot of flexibility in the setup of the clusters as more RAM can be added or removed by adding or removing storage servers while the coordinator will handle the data distribution. For better performance each cluster should be limited to a single datacenter, while difference clusters are deployed in different datacenters. This is because RamCloud server were developed for fast networks and as such don't do well when servers in the same cluster have high latency between themselves. This can lead ultimately to servers shutting down by themselves when the coordinator removes them from the cluster because of a false positive indicating that the server crashed. In this setup we only needed to increase the RamCloud client timeout for aborting RPCs to allow the clients to work across several datacenters. In reality that increase in the timeout is not necessary as RamCloud will attempt the write again if the coordinator indicates that the storage server is running and it will eventually succeed, however the increase leads to less retries, specially for bigger files that have to split in several objects smaller than 1MB.

4.2.4 TPM Attestation

The TPM attestation is implemented in two separate modules. One runs on the server and simulates the operations executed by the a real TPM, generating and writing the hashes to the PCRs. Trousers and TPM-Emulator are both running on the server and provide the TPM interface used for this. On the client side a separate modules is executed when the client starts which requests a quote to Trousers running on the middleware server. The quote is then verified against expected values. These values were obtained ahead of time with the middleware server executing in a trusted environment. If the values match operations are performed normally, otherwise the client might choose not to communicate with the server.

4.3 Transparent Integration with JCA

In this Section we will explain the implementation of our cryptographic provider as well as giving some examples of how to use it and compare it to some other encryption algorithms. In our implementation we followed the design principles and standards of the JCA reference guide so that using our cryptographic provider isn't significantly different than using any other cryptographic provider available. Despise that, some differences still exist due the difference in the CBIR algorithm compared to standard algorithms. We will explain those differences and show examples of how to work with them at the end of this Section.

4.3.1 Provider Implementation

The cryptographic provider has one major division in which we will focus: the CBIR implementation and the CBIR plus symmetric cipher implementation. This division allows a programmer to use only CBIR or use CBIR and encrypt the data with a symmetric encryption scheme as needed, simplifying it's use. We will start first with the CBIR implementation and then explain the addition of the symmetric cipher.

CBIR

To implement the CBIR algorithm we used the MAC engine. There are two classes implementing the CBIR algorithm that extend the MacSpi class: CBIRDense and CBIRSparse. The CBIRDense class implements the CBIR algorithm for images, while the CBIRSparse class implements it for text. The programmer must choose the correct class for the kind of data being used. Both classes will handle feature extraction through third party libraries, OpenCV[2] for images and Porter Stemmer for text [64]. OpenCV must be installed separately while Porter Stemmer is included in our provider. Using these classes is done like any other standard MAC class with a single difference. Since we don't know how many features will be extracted we can't give an accurate output size when calling getMacLength until all the

[2]OpenCV (http://opencv.org) is open source computer vision and machine learning software library. It possesses several algorithms to detect and recognize faces, identify objects, find similar images, among others. It's available with a BSD license and used extensively by companies, research groups and govermental bodies.

data is passed to the class instance through the update methods. If all the data was already provided then getMacLength will return the exact size of the output.

In addition to the CBIR algorithm we also implemented a parameter generator, parameter spec, key generator, secret key factory and key spec classes. This allows a programmer to configure an instance with different parameters as required. The available parameters for CBIRDense are m, $delta$, the feature detector and feature extractor type. The parameter m and the feature extractor type will impact the size of the matrices generated that will be used as key for the encryption process. Since the extractor type will impact the resulting features we use m to indicate the key size although in practice the CBIRDense key are two matrices, one of $m \times k$ dimensions and another of $m \times 1$ dimensions with k being a value determined by the feature extractor. Each of these parameters can be changed when creating an CBIRDParameterSpec instance. For CBIRSparse the available parameters are the key size and the HMAC algorithm used internally. A specific HMAC provider can also be selected. If the provider selected isn't found we fall back to default providers. Finding which provider is being used can be done by calling the getProvider method in the CBIRSParameterSpec class. By default we use HMac-SHA1 from the BouncyCastle provider if installed or from the default JVM provider if not. Keys can be generated automatically by the key generator and stored or read from files. The CBIRSparse key is a valid key for the HMAC instance used. For m and $delta$ the default values are 64 and 0.5 respectively, for the feature detector and feature extractor types the default algorithm is SURF. SURF (Speeded-Up Robust Features) is one of the several algorithms available in OpenCV for feature detection and extraction. Other algorithms might be choosen for specific requirements of feature detection if necessary.

CBIR with symmetric cipher

The CBIR with symmetric cipher classes join the CBIR algorithms and a symmetric encryption algorithm, allowing a programmer to encrypt the features and data with a single cipher instance. These classes extend the CipherSpi class and are used just like other cipher instances with two differences. Because the output of the cipher operation will include the CBIR output we can only provide an accurate output size after all the data is passed to the cipher object through the update methods. This also causes the output of the update methods to only being ordered relative to the symmetric cipher output. The final update operation will provide the initial contents of the final output. This difference doesn't occur when calling directly the doFinal methods in a single part operation.

Parameter generator and specs, key generator and specs and secret key factory classes are also implemented allowing for configuring both CBIR and the symmetric cipher. To configure a CBIR with symmetric cipher instance a CBIRCipherParameterSpec is used. This is an abstract class that is extended by CBIRDCipherParameterSpec and CBIRSCipherParameterSpec. Both classes are instantiated by providing a transformation for the symmetric cipher and a set of parameters optionally. The parameters can be for the CBIR component, the symmetric cipher component, or both. Both parameter components are instantiated as if using just their respective ciphers. They will be stored and passed for the respective

cipher objects when initializing. The transformation is a regular JCA transformation used to specific a symmetric cipher algorithm, mode and padding. By default we use AES in CTR mode with PKCS7Padding. Keys can be generated by the key generator in a similar way to other cipher algorithms. When using a key generator for CBIR with symmetric cipher both a CBIR key and a symmetric key cipher will be generated. Since there are actually two keys being generated instead of one there can be some problems when defining a key size. We solve this by assuming that if a specific key size is requested during initialization of the key generator it is applied to the symmetric cipher. To define a specific key size for the CBIR component then a parameter spec with the specific key size for both the CBIR and symmetric cipher must be created. By default we use a 256 bits key for AES. If the default values for the symmetric cipher are acceptable for the programmer defining a different CBIR key size is done by creating a CBIRParameterSpec with the desired key size and creating a CBIRCipherParameterSpec using the previous spec. Everything else will be created with the default values and the CBIRCipherParameterSpec object can be used to initialize the key generator.

We also support creating key by parts. Since a CBIR with symmetric cipher key is essentially the CBIR key and the symmetric cipher key together a programmer can use previously generated keys to create a CBIRCipherKeySpec which can be used as key for a CBIR with symmetric cipher instance. Those keys can be generated in any way the programmer wants as long they are instances of the required key types (CBIRKeySpec and SecretKey). This allows for example to use a repository in which the CBIR key is the same for all documents, but the symmetric key is different for each document, or for a set of documents.

4.3.2 Programming Environment

The following examples show how a programmer could use our cryptographic provider to leverage CBIR in an application. In algorithm 9 we show how a key can be generated and used to initialize a MAC instance of CBIR. We use the default parameters in this example and don't specify how the img variable is created as that is not important as long it holds the contents of an indexable image. Although this example refers to encryption of image features the code would be identical for encryption of text features. The only change would be on line 2 and line 4 where "CBIRD"would be replaced by "CBIRS". In algorithm 10 we show an example code of using CBIR with a symmetric cipher with default parameters. Similarly to the previous example, this code refers to an image processing. To use text data we would replace "CBIRDWithSymmetricCipher"on lines 2 and 4 for "CBIRSWithSymmetricCipher".

Because of the format of the cipher text produced by CBIR, a byte array is not the best way of representing it as it forces the programmer to be familiar with the internal formatting used in our cryptographic provider. We provide an utility class called CBIRCipherText to avoid this situation. A programmer can just pass encrypted_features or cipher_text as the constructor argument and this class will parse the cipher text, returning the results in a

```
1 byte[] img← image contents;
2 KeyGenerator key_gen = KeyGenerator.getInstance("CBIRD");
3 SecretKey key = key_gen.generateKey();
4 Mac cbir = Mac.getInstance("CBIRD");
5 cbir.init(key);
6 cbir.update(img);
7 byte[] encrypted_features = cbir.doFinal();
```

Algorithm 9: Encrypting image features with CBIR

```
1 byte[] img← image contents;
2 KeyGenerator key_gen = KeyGenerator.getInstance("CBIRDWithSymmetricCipher");
3 SecretKey key = key_gen.generateKey();
4 Cipher cipher = Cipher.getInstance("CBIRDWithSymmetricCipher");
5 cipher.init(Cipher.ENCRYPT_MODE, key);
6 byte[] cipher_text = cipher.doFinal(img);
```

Algorithm 10: Encrypting image features and data

more user friendly format, exemplified in algorithm 11.

```
1 CBIRCipherText obj = new CBIRCipherText(cipher_text);
2 float[][] encrypted_img_features = obj.getImgFeatures();
3 byte[] symmetric_cipher_text = obj.getCipherText();
```

Algorithm 11: Splitting encrypted image features and data

The programmer will now have easy access to the correct format of the encrypted features, and to the cipher text produced by the symmetric cipher algorithm. When decrypting only the symmetric_cipher_text should be passed for the cipher instance. A follow-up example for decrypting the image encrypted in algorithm 10 is shown on algorithm 12.

```
1 AlgorithmParameters parameters = cipher.getParameters();
2 Cipher decipher = Cipher.getInstance("CBIRDWithSymmetricCipher");
3 decipher.init(Cipher.DECRYPT_MODE, key, params);
4 byte[] plain_text_img = cipher.doFinal(symmetric_cipher_text);
```

Algorithm 12: Decrypting an encrypted image

One thing to take note on the last example: on line 1 we obtain the parameters used by the cipher instance. This is not necessary if the parameters were generated by the programmer. In the case of using default parameters this can be used to obtain the IV used during the cipher process. The method getIV() could also be used but it returns a byte array which would then need to be turned into an IVParameterSpec to be used in another cipher instance. This IV is generated automatically unless the programmer passes their own IV when initializing the cipher for encryption. To use their own IV a programmer needs to create an IVParameterSpec and then pass that object to the init method of the cipher.

4.4 Deployed Environment for Evaluation Test benches

To evaluate the performance of our system we deployed it in several environments: local environment, a single datacenter and several datacenters. In this Section we describe each of these environments and how the system was deployed in each of them. In all the environments the machine running the client proxy was a i7-4700MQ @2.4GHz with 4 cores and hyper-threading and running Kali with kernel 4.6.0 and 12GBs of RAM. We used Amazon EC2 and S3 for all environments, with the middleware server deployed in the Frankfurt region. Because RamCloud uses busy waiting special care was taken to make sure all instances had enough credits to perform at full capacity for the duration of the tests.

In these evaluations we chose to use Amazon datacenters to have a comparison base between DepSky and RamCloud which would be harder to evaluate if using datacenters from different providers when comparing both storages. For RamCloud we also used a single storage server. While on a real use case several storage would need to be used for the storage capacity this doesn't affect the measurements as the internal replication from RamCloud wouldn't be necessary and can be asynchronous if used. Since we abstract a storage cloud into a cluster, replicating a file inside the same cluster would be equivalent at writing the same file more than once in a storage cloud using Depsky, which would be redundant. Persistence isn't affected as files are fragmented over several clusters and we can reconstruct the file from the remaining clusters.

4.4.1 Local environment

For the local environment test we placed both the middleware server and the storage clouds in a single machine, represented in figure 4.1. This environment has the objective of minimizing latency so both the client and the server machines are connected on an isolated network. Because of the overhead of having four coordinators running and the memory requirements the RamCloud backend has a difference compared to the other environments. Instead of having four coordinators, each responsible for a cluster, we have a single coordinator, and abstract the storage clouds as tables instead of clusters. This reduces the overhead caused by the busy waiting used by RamCloud, which creates less impact on the indexing operations. In this situation we can reduce the number of coordinators and expect similar results to what it would be as if the deployment was the same as the other environments because if all coordinators were to be running in the same machine, having the client connect to coordinator 1 is no different than connecting to coordinator 2. The server used in this environment is an AMD A6-6310 APU @1000MHz with 4 cores running Ubuntu and with 8Gbs RAM.

4.4.2 Single Datacenter Multi Cloud Environment

For test bench 2, represented in figure 4.2 we used clouds in the same datacenter as the middleware server. The middleware server machine was a T2.large instance while the

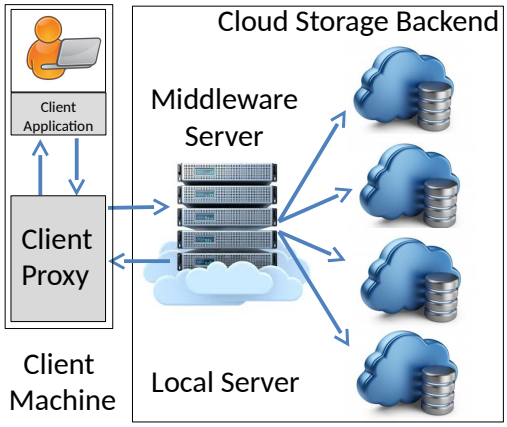


Figure 4.1: Test bench 1

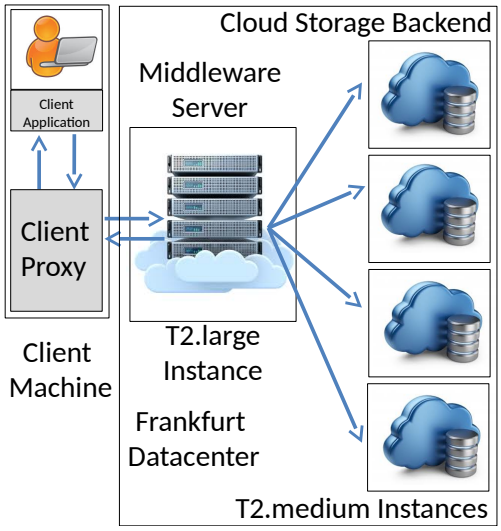


Figure 4.2: Test bench 2

RamCloud storage server and coordinators were deployed in T2.medium instances. For Depsky we created different buckets all in the same region.

4.4.3 Multi Datacenter Multi Cloud Environment

For test bench 3, represented in figure 4.3 we used cloud in different datacenters. The middleware server and one storage cloud were deployed in the Frankfurt region, while the other storage clouds were deployed in Ireland, Oregon and North Virginia. For DepSky buckets we created one bucket in each of the regions. For RamCloud the deployment is more complex as the storage servers and coordinators provide their IP to the ZooKeeper instance and is that IP that the client will use. However the public IP of the T2 instances isn't actually connected to the instances and belongs instead to a router that allows instances to communicate to addresses outside their VPC. RamCloud only allows IPs that are attached to the machine were it's running, preventing the use of the public IP to receive connections. To solve this and allow the client to communicate with the coordinators and the storage servers we use IPSec to create tunnels between the different VPCs in which the instances are deployed.

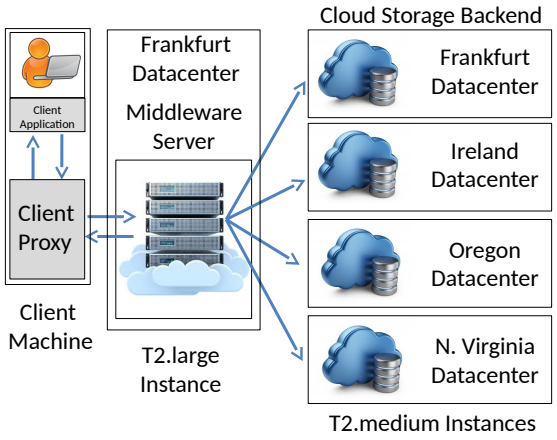


Figure 4.3: Test bench 3

4.5 Implementation Effort

In this last section we summarize some figures related to the implementation effort as describe in the previous sections. The project has 12724 lines of code, with 1881 of those being reused from a previous implementation of CBIR. This includes all components of the project but not it's dependencies. For any of the dependencies required only the code required to use them was included. This project was mainly written in C/C++ for the middleware server and in Java for the client proxy. Overall there is 6822 lines of code in C/C++ and 5902 in Java, with the middlware server creating a package with 16 classes and the client proxy, together with the cryptographic provider, creating a package with 47 Java classes.

To provide a metric of effort involved COCOMOII was used to provide a reference. This model is more suited for a team project with on-going maintenance and so it's estimations can be inflated to the effort involved however it is still an interesting evaluation for the implementation effort. In figure 4.4 we present the settings used and the respective results obtained with the use of an online tool available at [65]. The parameters are explained in great detail in [66] and as such will not be explained here.

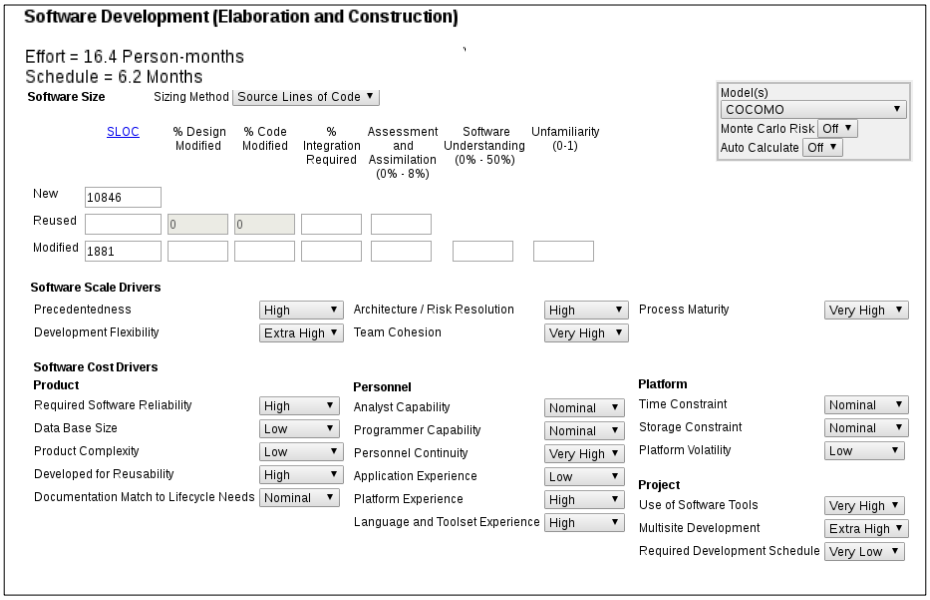


Figure 4.4: COCOMOII Metrics

Using these parameters the effort was measured at 16.4 person-month with a schedule of 6.2 months. While this may seem too much it is important to note that this model assumes a project with more than one person working on it and with several versions that need to be compatible. The integration of the work done by different people together with the tests required to make that all funcionalities are working correctly even with previous versions of the project would increase the time taken which is reflected on this evaluation. In this project the integration of several dependencies was done once for each dependency and while care was taken to make the system easy to extend with new cryptographic algorithms or new backend types there wasn't a need to test new changes with previous versions.

CHAPTER 5

EXPERIMENTAL EVALUATION AND ANALYSIS

We present in this chapter the results of the experimental evaluation of the proposed system. As previously addressed, the test bench environments are built on different configurations and specific deployments. The assessment results presented next are structured according to the used test bench environments as initially characterized in Section 4. In all the test benches we deployed all the system components in a machine on a single docker instance.

5.1 Test Bench 1 - Local Base Environment

Based on the test bench 1 deployment we evaluated the following aspects of our implementation: (1) cost of encrypting data with CBIR as opposed to use only AES, (2) cost of the setup (encryption, upload and indexing of the data), (3) searching for uploaded documents, (4) retrieval of documents with cache on the middleware server and (5) retrieval of documents without cache in the middleware server. In each one of the following sub-sections we focus on each evaluated aspect presenting experimental test behaviors, presenting the achieved results and discussing the observed results.

5.1.1 Cost of Using CBIR

On tables 5.1 and 5.2 we show the comparison between CBIR and AES for one document. Those values are the average processing time for 1000 different documents, resulting from 5 different observations, in milliseconds. For the CBIR column we show how long it took for the feature extraction, indexing and encryption, which were measured in the cryptographic provider. The total row is the time measured in the client application. For this test we used a single doFinal operation with all the data. For the AES column we only show the total value as we used the BouncyCastle. implementation.

It can be seen that the CBIR process is much more expensive than standard symmetric encryption, however there is also more processing involved. Extracting and computing indexable features takes 228.262 ms, which is 44.45 % of the total time for CBIR Dense and 0.084 ms for CBIR Sparse, which is 56.76 % of the total time. This process also causes data expansion on most documents. In our dataset the plain text size of the 1000 images is 111.561 MB and the textual data is 83.147 KB, while the features size is 321.205 MB for the images and 181.504 KB for the text. This cause the input data for encryption on the CBIR to be more than the input data for AES which contributes to the difference in times.

	CBIR Dense	AES
Feature Extraction	65.705	0
Indexing	162.557	0
Encryption	285.217	0
Total	513.551	1.407

Table 5.1: Comparison of CBIR Dense and AES for one image in ms

	CBIR Sparse	AES
Feature Extraction	0.084	0
Indexing	0	0
Encryption	0.043	0
Total	0.148	0.012

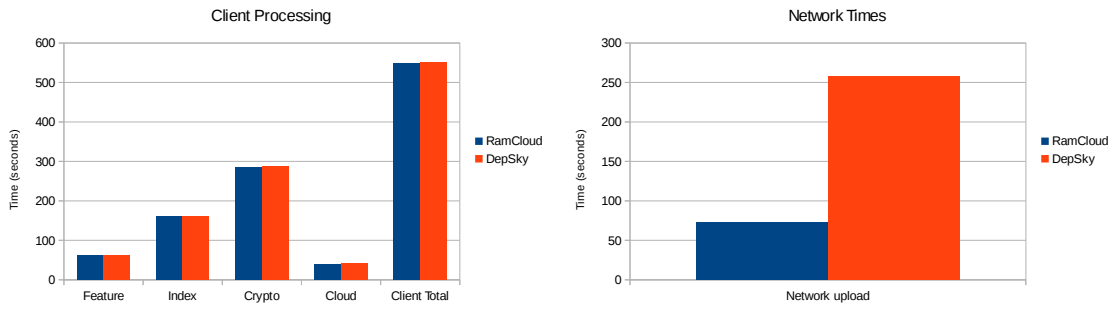
Table 5.2: Comparison of CBIR Sparse and AES for one text document in ms

5.1.2 Cost of Setup

Documents need to be uploaded and indexed before users can retrieve and search them. We refer as those two operations the setup of the system. We measured the cost of uploading and doing the first indexing operation on all test benches to compare Depsky with RamCloud and to see the effect that higher latencies have on each of the backends. We present next the results of uploading and indexing in a local environment.

Upload

Figure 5.1 presents the results of the upload test. In figure 5.1a feature represents feature extraction, index represents the feature processing, cryptograpic represents encryption time, cloud represents the time the client spent uploading and waiting for the middleware server and client total represents the total time measured by the client. There are no significant difference between DepSky or RamCloud in any of the measurements. This is expected for the feature, index and cryptographic assessement as this processing doesn't require any interaction with the backend storage. However one could expect a noticeable difference in the cloud measurement as latencies are very low which allows RamCloud to make more use of the speed of RAM storage compared to disk storage. In figure 5.1b we see the time spent by the middleware server in uploading data to the storage backend. While the client time is



a Client times

b Server times

Figure 5.1: Upload times for test bench 1

very similar for both RamCloud and DepSky, the upload time from the middleware server is very different. RamCloud took 73.251 seconds while DepSky took 257.503 seconds to store all 1000 images, a difference of 351.54 %. The reason for this difference not being noticed in the client side is the write protocol of our system and the time it takes to process one image on the client proxy. Once the client receives a confirmation from the middleware server that the document was received the upload operation is completed. This makes it so that only the upload time until the middleware server is noticed by the client. The middleware server uploads the document to the storage backend after sending the client the confirmation that the document was received. Another factor is that the client takes on average 506 ms processing each document. Looking at the upload values of the middleware we see that RamCloud takes on average 73 ms per document while DepSky takes 257 ms. Both values are less than the time the client requires, making the client the bottleneck on this test. If that was not the case then it would be possible that the client would see some difference between DepSky and RamCloud.

The difference between the times measured by RamCloud and Depsky can be explained by the test bench setup. In this setup we are not using clouds but a single local server. To simulate the clouds we used the local driver provided with the DepSky implementation, which uses the server disk to store the data. Since the local driver simulates 4 clouds, all 4 fragments had to be written in the same disk, causing a significant overhead. While RamCloud also had some overheads, they were not as significant as the processing done in the middleware server during upload is not very heavy and although it also had to store the 4 fragments in RAM the write in RAM was faster and less affected by that overhead.

Training and Indexing

The cost of the training and indexing operation are presented in figure 5.2. In this figure train represents the training phase and the storage of the generated codebook, index represents the indexing time, network feature represents the time reading or writing the features to the storage backend, network index represents the time spent writing the indexes to the storage backend and network upload and download represent the time that was spent on every write or read during these phases. Because the training phase takes much longer

Figure 5.2: Measurements of the training and indexing phases in test bench 1

than the remaining phases the scale presented is logarithmic. Table 5.3 shows the values in seconds as well as the variation in percentage of RamCloud compared to DepSky. We can

	DepSky	RamCloud	Variation %
Train	1293.863	1639.812	26.738
Index	38.038	57.180	50.324
Network feature	25.949	32.821	26.485
Network index	2.343	2.459	4.914
Network upload	16.784	9.583	-42.907
Network download	0.229	0.145	-36.703

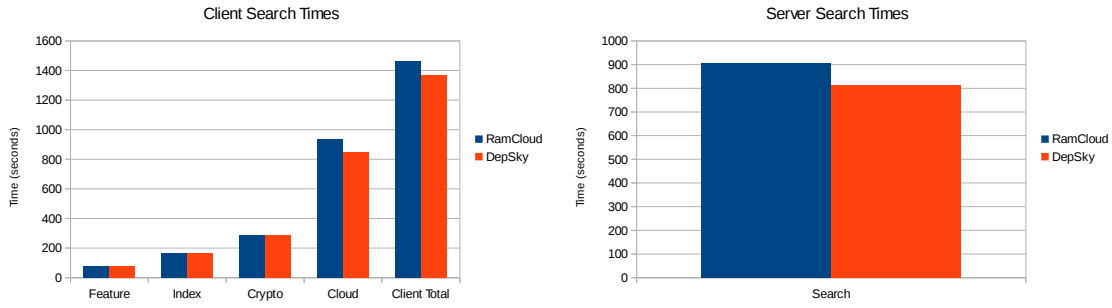
Table 5.3: Comparison for train and index times between DepSky and RamCloud for test bench 1

see that DepSky has better results in all but the time spent writing or reading. The network feature and network index times were measured outside the storage module and show how long the function calls took, while the network upload and network download times were measured inside the storage module and represent the time since the first request sent until enough replies are received to conclude the write or read. This difference in times is because the storage operations are more expensive for DepSky, however it doesn't have any processing overhead. RamCloud uses busy waiting in both the coordinator and the storage server, making all processing operations slower. Both of the network feature and network index operations make use of our second type of writes for big files which causes files to be split and buffered before being sent to the storage backend. This pre-processing together with the overhead of busy waiting makes DepSky faster than RamCloud, despise the slower read and write operations. This overhead is even more noticeable for the training phase, in which RamCloud is 26.738 % slower, but that difference represents nearly 346 seconds. For the indexing phase RamCloud is 50.324 % slower, although that translates to a 19 second difference, much less than the difference for the training phase. One thing to note is the the docker container runs it's own scheduler. When we split the components in two containers,

one running the coordinator and the storage server and another running the middleware server the processing times got considerably worse. Checking the processes while in the training phase we saw that with two containers the coordinator and storage server were among the top programs with 100 % CPU utilization each with the remaining being left for the middleware server. When running only one container the CPU utilization of the coordinator and storage server decreased, making the middleware server the process with most CPU time and reducing the time that it took for the training and indexing.

5.1.3 Cost of Searching

The results for the search evaluation are presented in figure 5.3. We can see that although



a Client times

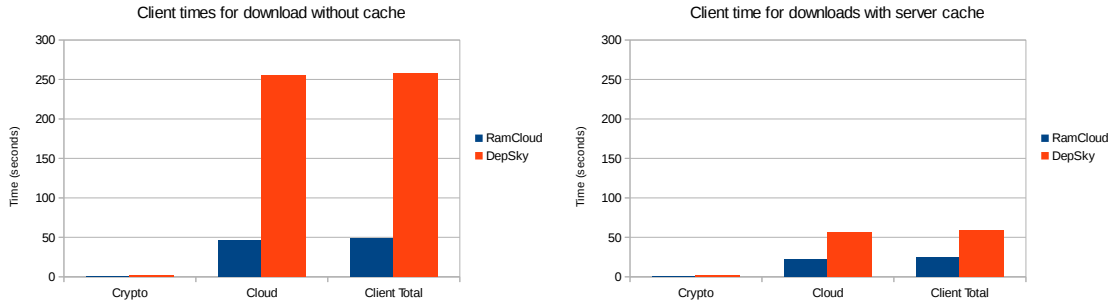
b Server times

Figure 5.3: Search times for test bench 1

the search operation doesn't make use of the storage backend the RamCloud version is slower than the version running DepSky. That difference comes from the time the client spends waiting for the server to send the search results and we see that the middleware server spends more time searching when using RamCloud than when using DepSky. This difference is caused by the coordinator and storage server overhead. Since there is no writes or reads the overhead from DepSky doesn't affect this operation, however the busy waiting method used by the two servers slows down the search. Since we used a single threaded client the multi thread factor of the middleware server that allowed the client to not notice the difference between backends isn't applied here as the client will not start searching for the next image until it receives the results for the previous one.

5.1.4 Cost of Retrieval

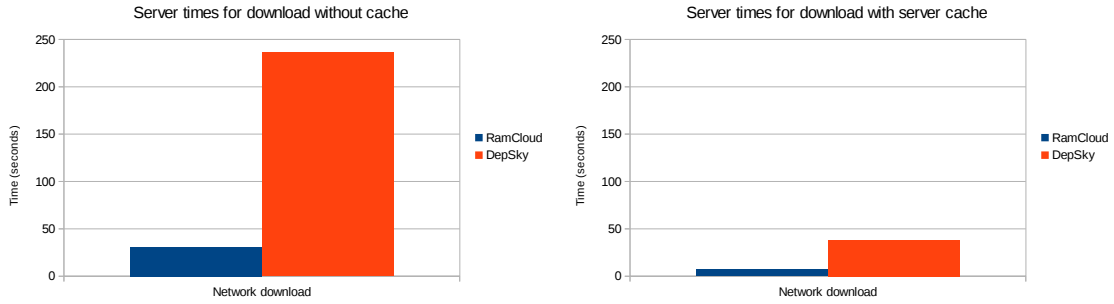
In figure 5.4 we can see the time measurements on the client side and in figure 5.5 the measurements on the server side. We don't show the feature and index measurements on the client side as they aren't used for downloads. We can see in 5.4a that downloads are much faster with RamCloud than with DepSky. Using DepSky is 428.457 % slower than a RamCloud on the client side while on the server side, shown in 5.5a DepSky takes 665.167 % more time to read the files. The extra overhead from the RamCloud processing



a Client times without cache

b Client times with cache

Figure 5.4: Download times for test bench 1 on the client side



a Server times without cache

b Server times with cache

Figure 5.5: Download times for test bench 1 on the server side

doesn't offset the slower reads from DepSky. For the cache test we assumed an hit rate of approximately 80 %. We can see those results in figures 5.4b and 5.5b. Comparatively to the results without cache, RamCloud takes 22.6 % of the time to retrieve the files while DepSky takes 16.139 % of the time without cache. On the client side we can see that the RamCloud time is 50.012 % and the DepSky time is 22.582 % of the time without cache. This makes DepSky much closer to RamCloud, being now 138.614 % slower. It's obvious from these values that DepSky benefits much more from the cache than RamCloud. This makes sense as the cache avoids the slower reads from disk for DepSky as well as the reconstruction of the file, while for RamCloud it only avoids the reconstruction since the data will be read from memory when retrieving from the cache or from the storage.

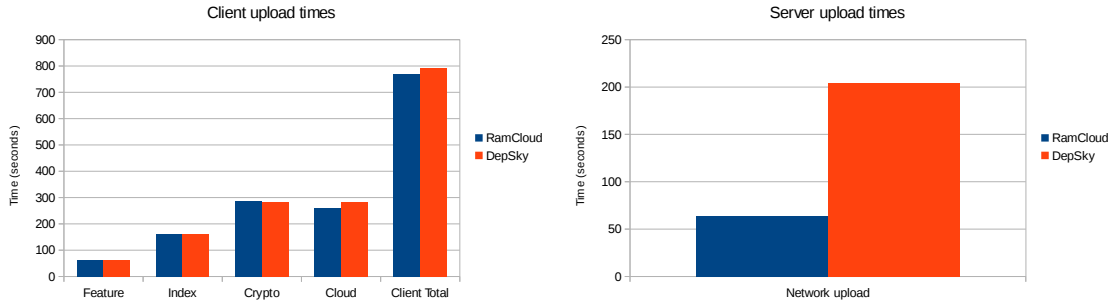
5.2 Test Bench 2 - Multi Cloud in the Same Datacenter

In this Section we focus the evaluation on the test bench 2 deployment, from which we performed the following experimental observations: (1) cost of setup, (2) searching, (3) retrieval with cache in the middleware server, (4) retrieval without cache on the middleware server. In the next sub-sections we present for each observation the experimental test behaviors, the observed results and the interpretation and analysis of those results.

5.2.1 Cost of Setup

Upload

Figure 5.6 shows the results for the uploads in test bench 2. While the feature, index,



a Client times

b Server times

Figure 5.6: Upload times for test bench 2

and cryptographic values show the expected results with the same values in both DepSky and RamCloud, the cloud and client total values show DepSky being slower than RamCloud. With the increase in latency between the client and the middleware server we expected these values to be closer as the higher latency makes the faster writes of RamCloud less noticeable. One possible reason when looking at figure 5.6b is that the slower uploads times seen in the middleware server with DepSky might be the bottleneck which would increase the time that the client spends on uploads as it waits for previous uploads to complete and free a thread. However that is not the case as DepSky spends approximately 200 seconds uploading, which gives an average of 200 ms per document. On the client side we can see that the processing of a document alone takes more than 200 ms and so DepSky can't be the bottleneck. The reason for the extra time measured on the client side is variation in the latency of the connection from the client to the middleware server. The standard deviation for the cloud value in DepSky is 36.544 seconds and for RamCloud is 6.585 seconds. Furthermore, the minimum value measured for DepSky in the cloud measurement is 252.826 seconds. That value is actually lower than the minimum of RamCloud at 254.298 seconds. We can assume then that the difference noticed in the client was not caused by a difference in the storage backend as the connection between the client and the server is the bottleneck in the upload. The difference in the upload times measured in the middleware server are expected based on the results obtained in test bench 1. While we moved the middleware server and storage to the cloud, all components are in the same datacenter and the latency from the middleware server to any of the storage clouds is less than 1 ms, which is very close to test bench 1 but without the extra overhead. In this environment the faster writes and reads in RAM from RamCloud can still be noticeable when comparing to DepSky. It is interesting to see however that while RamCloud takes slightly longer, DepSky is actually faster in this test bench than in test bench 1. This makes sense when considering that latencies for the middleware server

are very close in both test benches, but in test bench 2 DepSky can write all fragments in parallel in different clouds instead of in sequence in a single disk. RamCloud would not benefit as much from this change as writing in memory is already a very fast operation.

Index

In figure 5.7 we show the results for the training and index phases. In table 5.4 we show the exact values as the scale presented in the figure is logarithmic. We can see

Figure 5.7: Measurements of the training and indexing phases in test bench 2

	DepSky	RamCloud	Variation %
Train	1016.462	992.785	-2.329
Index	33.738	32.731	-2.986
Network feature	22.028	26.761	21.489
Network index	1.280	1.040	-18.698
Network upload	20.768	12.530	-39.666
Network download	0.184	0.120	-34.655

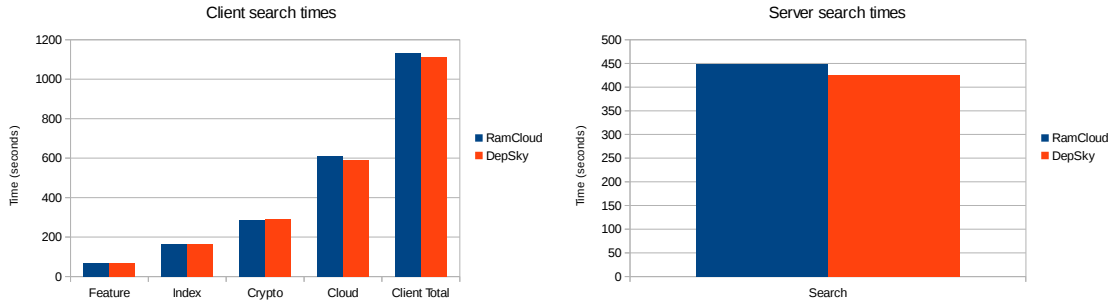
Table 5.4: Comparison for train and index times between DepSky and RamCloud in test bench 2

that the training and indexing have similar values between DepSky and RamCloud. The difference between the indexing is approximately 1 second, with DepSky being 2.986 % slower than RamCloud. Although we expected the same value on both the results are close enough that is possible that it was was caused by the scheduling of the operations on the system. For the training phase it was expected that RamCloud was faster based on the results seen on the upload test since the training phase also includes storing the codebook generated in the storage backend and in that operation RamCloud would be faster. For the network feature we expected RamCloud to perform better, however DepSky is 21.489 % faster than RamCloud in that operation. Although unexpected it's not surprising as the features generated a file of approximately 600MB. We split that file for both DepSky and RamCloud before the fragmentation with erasure codes, with DepSky handling the file segments directly. RamCloud doesn't support files bigger than 1MB and so we have to further split the file into 1MB pieces after the fragmentation. We make use of multi write

rpcs to write all pieces using a single RPC, however the extra processing makes this write slower. When looking at the network index result we see that RamCloud is again faster than DepSky, with the index file being much smaller with 6MB. Although we use the same kind of write for both features and indexes the latter aren't split. Looking at the network upload and network download times we see that DepSky is 39.666 % and 34.655 % slower than RamCloud respectively. Comparing this variation with the ones seen on test bench 1 we see that it decreased. This is expected as although RamCloud have faster writes and reads, as latency increases the more similar times DepSky and RamCloud will have.

5.2.2 Cost of Searching

In figure 5.8 we present the results for the search operations. There is a slight difference



a Client times

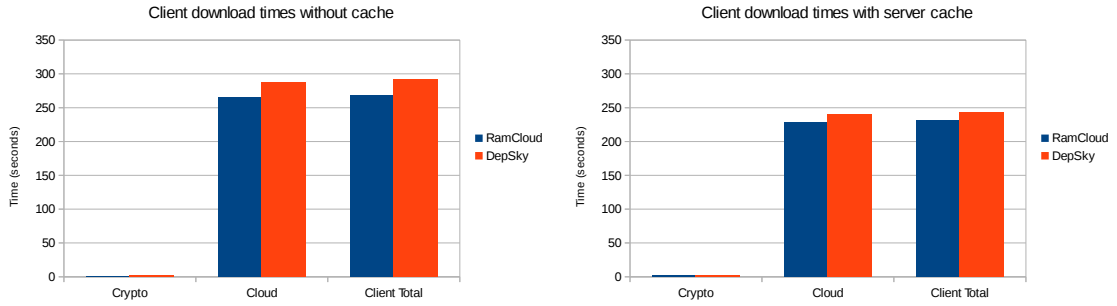
b Server times

Figure 5.8: Search times for test bench 2

between RamCloud and DepSky, with RamCloud being 1.644 % slower than DepSky, and with the search in the middleware server being 5.439 % slower. We expected similar results for both kind of backends since the backends aren't actually used in this operation and although there is a slight difference it is due to the scheduling and processing time of the system. The results for both DepSky and RamCloud varied a lot between themselves, with a standard deviation of 26.592 seconds for DepSky and 32.068 seconds for RamCloud. That is more than the difference between the average of the client total (18.305 seconds) and the search time (23.120 seconds). Pinpoint the exact cause of this variance is not direct as it can be caused by the scheduler of the docker container or the system or a combination of both as noted on test bench 1.

5.2.3 Cost of Retrieval with Middleware

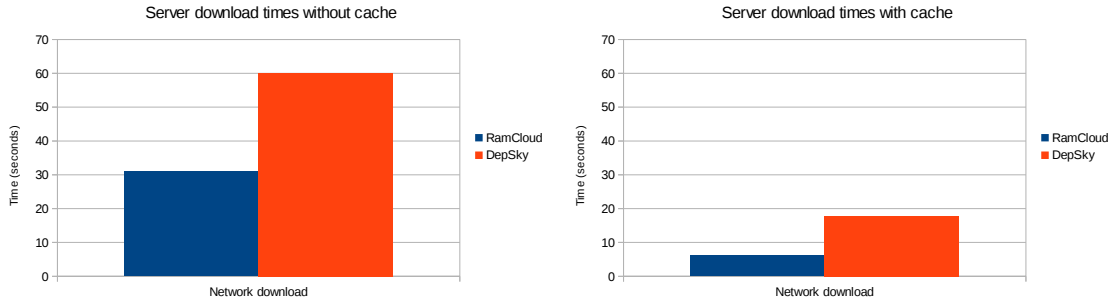
In figures 5.9a and 5.9b we present the results for the download test on the client side. Figures 5.10a and 5.10b show the results measured on the server side. Comparing DepSky with RamCloud we see that RamCloud is faster by 8.048 % when measured from the client side, but spends only 51.544 % of the time of DepSky in the middleware server. Although it might seem that RamCloud is slower after download the file from the storage backend when



a Client times without cache

b Client times with cache

Figure 5.9: Download times for test bench 2 on the client side



a Server times without cache

b Server times with cache

Figure 5.10: Download times for test bench 2 on the server side

looking at the values we see that the difference measured between the times in the client side is 23.465 seconds and in the middleware server is 29.141 seconds, with a standard deviation of 23.381 seconds for the cloud time in RamCloud and 4.934 seconds in DepSky. If we look at the values with cache we can see that RamCloud is 13.727 % faster while DepSky is 16.443 % faster compared to the no cache version. In the middleware server we can see a reduction of 80.220 % on the download time for RamCloud and 70.504 %. This represents a reduction of 24 seconds for RamCloud and 42 seconds for DepSky which is expected. Given that DepSky takes longer to retrieve the documents it will benefit more from the cache as it will save more time.

When we compare the benefits of the cache on the client side on this test bench with the ones obtained in test bench 1 they appear to be low. This is explained by the increase in the distance between the client and the middleware server. Using the values with cache as they are more constant we can calculate that with a 100 % hit rate on the cache the client would still take 225 seconds to download all documents, which leads to 0.225 seconds per document. Using the values obtained in the middleware server without cache we can assume 0.03 seconds per document for RamCloud and 0.06 seconds for DepSky, both much faster than the download between the client and middleware server, making the connection between the client and the middleware server the bottleneck.

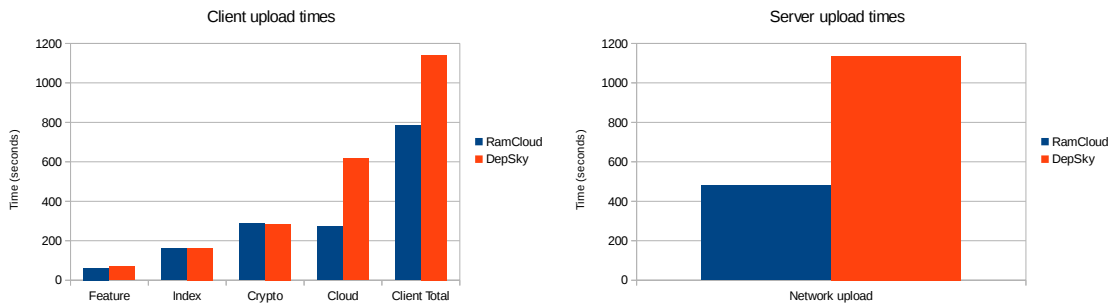
5.3 Test Bench 3 - Multi Cloud in Several Datacenters

This Section is dedicated to the evaluations conducted with the test bench 3 deployment. Based on this test bench we conducted the following experimental tests: (1) cost of setup, (2) retrieval with cache on the middleware server and client, (3) retrieval with cache on the middleware server and without cache on the client, (4) retrieval without cache on the middleware server and with cache on the client, and (5) retrieval without cache on the middleware server or on the client. For each experimental purpose we present in the next sub-sections the experimental environment, the obtained results and the critical analysis of the observed results.

5.3.1 Cost of Setup

Upload

In figure 5.11 we show the results for the upload of 1000 images when the storage clouds are in different datacenters. The feature, index and encryption measurements are similar,



a Server times without cache

b Server times with cache

Figure 5.11: Upload times for test bench 3

however the cloud and total values present a considerable difference between DepSky and RamCloud. DepSky takes 357 seconds longer, making it 45.631 % slower than RamCloud. When looking at the server times we can see that the difference is bigger, with DepSky taking 658 seconds longer than RamCloud. Looking at the upload time on the server for RamCloud we can see that the bottleneck is the client as it takes 509 ms to get one document ready to upload, while in the server RamCloud only takes 478 ms to perform the upload. This explains why the difference seem in the middleware server between RamCloud and DepSky is not the same as the one seem by the client but it doesn't explain the difference between DepSky and RamCloud. With the higher latencies we expected the results to be closer as latency would be nearly the same for both storage backends. To explain this difference is necessary to look at the implementation and traffic generated by DepSky as well as the test bench conditions.

For the upload tests we always started with an empty storage. This was to avoid the overhead of reading files which could start introducing differences from the start and to

make sure both DepSky and RamCloud would start in the same conditions. While both DepSky and RamCloud follow the same high level protocol when writing data consisting in 3 steps: read current metadata, write new data and write new metadata, they differ greatly in the implementation. The first step of reading metadata is where the differences start. To move to the write data step is necessary 3 storage clouds finish the read so that there is a quorum about the current metadata. RamCloud issues a request to each of the storage servers and when it receives 3 answers the step is finished and the write data starts. DepSky, like RamCloud, issues a request to each storage cloud, but doesn't necessarily move to the next step once 3 answers are received. In the case that the a file doesn't exist DepSky assumes an error and triggers an exception causing a retry of the request, for another 3 times if all fail. Since we start with a empty storage, this behavior is triggered on every upload and the 3 extra operations on each cloud start accumulating a considerable difference. Although in implementation RamCloud is more complex as it requires extra operations to deal with the split of the files in 1MB segments we make use of multi operation RPCs as well asynchronous RPCs, which allow us to invoke several RPCs without waiting for the previous one to finish, making sure they terminate only after issuing all the requests needed. This implementation allows RamCloud to perform all operations in the same time as it would to execute 3 operations, while waiting for the previous one to finish before starting the new one.

Another aspect which delays DepSky, although not related to it's implementation, is the protocol of the write operation for Amazon S3. Each write operation is composed of 2 operations, a first initial request followed by the data after the storage cloud sends the response to the first request. Although SSL is used on these requests and we can't directly see the data on that first request, it is possible that it corresponds to the header of the request. Looking at Amazon S3 API we can see that the PUT request allows an application to send the request header before sending the request body so that data is not sent if the upload won't succeed. While this allows to save on network traffic, it delays the write by one RTT as a response from the server is required before starting the upload. For an upload operation the delay is actually 2 RTTs as there is a delay for writing the data and then another one for writing the metadata. On the previous test benches this was not noticed as the latency between the middleware server was small, not reaching 1 ms between middleware server and any of the storage clouds. However in this test bench latencies between the middleware server and the storage clouds can reach up to 161 ms for the slowest storage cloud. Since the read operation requires a quorum we can assume that, in general, the latency of that cloud won't impact the reads unless one of the other cloud fails. However, this assumption only reduces the latency of the slowest used cloud to approximately 92 ms. It is possible that by using a different storage cloud, for example Microsoft Azure or Google Cloud Storage, this extra request before each write wouldn't happen, reducing the time for the uploads.

Index

Figure 5.12 shows the results for the training and indexing phases and the exact values are presented in table 5.5. The training and indexing phases are very similar in both Dep-

Figure 5.12: Measurements of the training and indexing phase in test bench 3

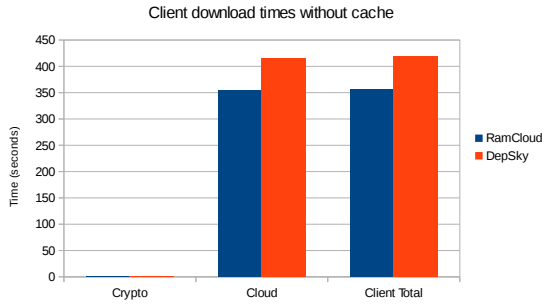
	DepSky	RamCloud	Variation %
Train	1036.207	1037.608	0.135
Index	34.016	35.614	4.699
Network feature	87.772	78.630	-10.415
Network index	3.697	3.843	3.945
Network upload	71.215	68.117	-4.350
Network download	1.626	0.899	-44.706

Table 5.5: Comparison for train and index times between DepSky and RamCloud in test bench 3

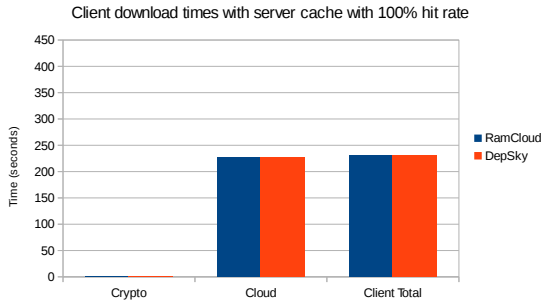
Sky and RamCloud. Although that is expected for the indexing, we would expect DepSky to perform worse in the training phase as that includes the upload of the codebook. The same happens in network index, in which DepSky is faster than RamCloud by 1.598 seconds. The values for the network upload are also very close. The reason for this is in the number of uploads and amount of data sent. The codebook and indexes are small files in which only one upload is required, but the features require 4 uploads per cloud. With more uploads the overhead of the writes in DepSky becomes more noticeable. Another thing to take into consideration is the latency between the clouds. Because this are the results of 5 tests a variation in only a few tests might skew the results. The standard deviation for the network index value is only 0.202 seconds, but for RamCloud is 1.271 seconds. For the network feature the standard deviation is 19.112 seconds for DepSky and 8.979 seconds for RamCloud. For the network upload the standard deviations are 9.068 seconds for DepSky and 8.621 seconds for RamCloud. This seems to indicate that values for DepSky and RamCloud would tend to be closer to each other with more tests performed which makes sense as the number of uploads performed is quite small and the difference between the two version should be minimal.

5.3.2 Cost of Retrieval

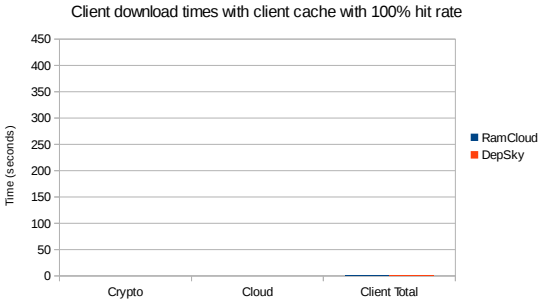
In figure 5.13 we present the results for the download evaluation on the client side without cache and with cache with 100 % hit rate on both the client and the server. The cache on



a Client times without cache



b Client times with server cache with 100 % hit rate

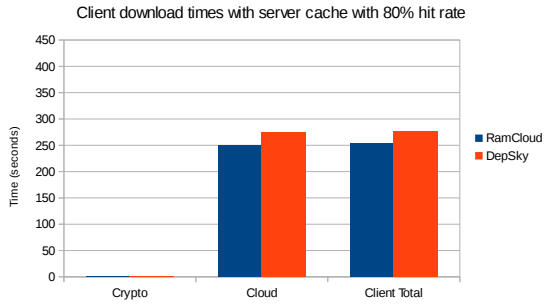


c Client times with client cache with 100 % hit rate

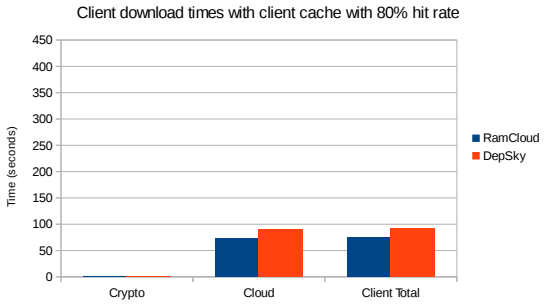
Figure 5.13: Client download times for test bench 3 without cache and with cache with 100 % hit rate

the client side is the most efficient in this test as it avoids all processing to decipher the documents on the client side as well as all processing and download from the middleware server. The server cache although avoids all processing related to the storage backend such as download, integrity check and reconstruction is less efficient since the client still needs to download the documents from the middleware server. We could see that in test bench 1 the server had more impact as the latency between the client and the middleware server was much smaller. Although the results with the cache are the expected, in figure 5.13a we can see that DepSky is 17.174 % slower than RamCloud, a difference of 61 seconds. Although in previous test benches RamCloud was faster the latency between every cloud was also very small. We expected that the higher latencies from this test bench would mitigate that difference making both DepSky and RamCloud similar in performance. Looking at figures 5.13a and 5.15a we see that it increased from test bench 2, with the difference going from 29 seconds to 68 seconds in the times measured in the middleware server.

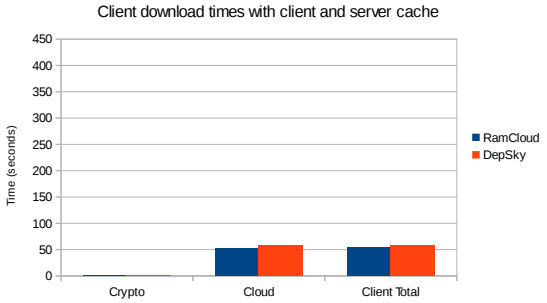
We used Wireshark to capture the traffic in the middleware server for 10 requests and evaluated the times of packet arrivals in comparison with the expected results from figure 5.15a. From that capture we made an average of 199 ms since the first packet of an upload was sent until the last necessary packet to conclude the upload was received for DepSky and 117 ms for RamCloud. We expected 193 ms for DepSky and 125 ms for RamCloud based on the values measured during the tests. Although we only used 10 requests, as opposed to



a Client times with server cache with 80 % hit rate



b Client times with client cache with 80 % hit rate

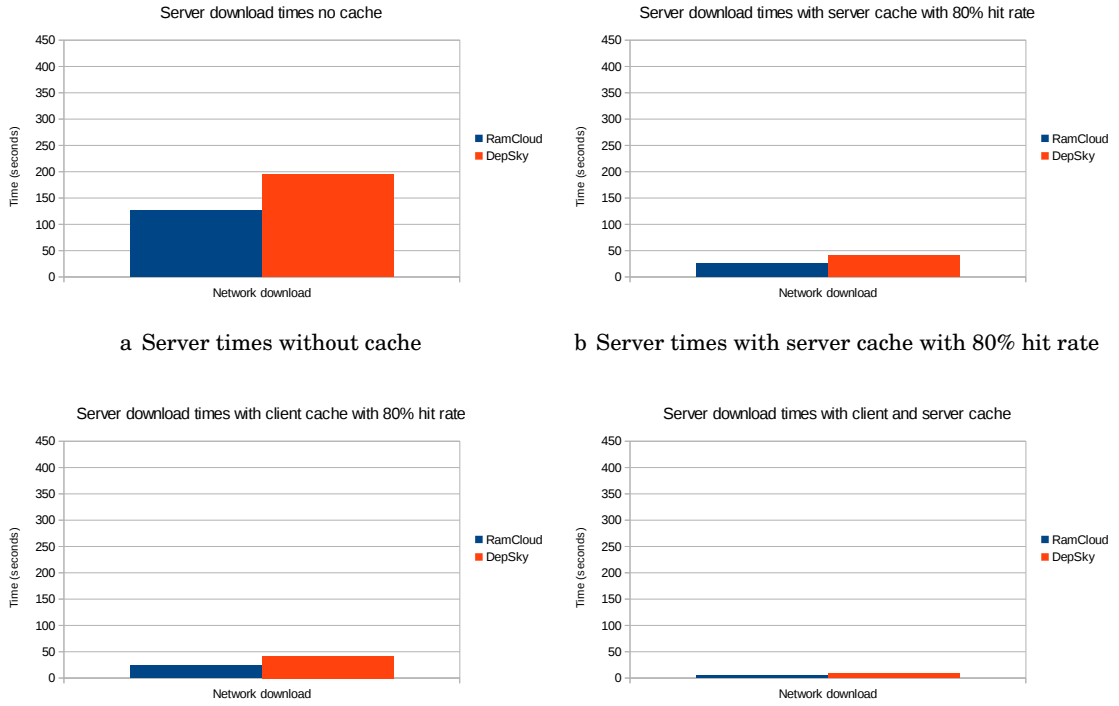


c Client times with client cache and server with 80 % hit rate each

Figure 5.14: Client download times for test bench 3 with cache with 80 % hit rate

the 1000 made during the download tests, values are very close to the ones expected, with the standard deviation being 69 ms for DepSky and 12 ms for RamCloud in the Wireshark capture. Although not directly comparable the standard deviation on the tests was 6.455 seconds for DepSky and 5.506 seconds for RamCloud. While this seems to support that even with higher latencies RamCloud is faster we also measured the time that both DepSky take on average to read the metadata and the fragments on individual clouds. These times were measured since the first packet with the request was sent until the last packet with data was received. For DepSky we got an average of 20 ms for reading the fragments on the closest cloud to the middleware and 125 ms on the second closest cloud while for RamCloud we got 12 ms for the closest cloud and 69 ms for the second closest cloud. We didn't measured this values for the remaining clouds as although fragments were read in some of the requests they were never used to complete the request and didn't directly impact the time for the retrieval of the file. There could be some influence in further retrievals as in particular request the metadata stored in the slowest cloud was used because one of the faster clouds was still busy retrieving the fragment for a previous request that had already been completed, however in our capture this only affected the one of the following requests. In the request following that one that cloud had already completed every pending request. During our Wireshark capture this only happened with DepSky, although it would be possible to happen with RamCloud as well. For the metadata we got 18 ms, 32 ms, 115 ms and 155 ms averages for DepSky with 13 ms, 12 ms, 3 ms and 25 ms of standard deviation

and 7 ms, 30 ms, 104 ms, and 172 ms averages for RamCloud with 4 ms, 5 ms, 4 ms and 5 ms of standard deviation. Although the slowest cloud on RamCloud is slower than the equivalent cloud in DepSky its responses were never used to complete a request. These values support the test results that show RamCloud being faster than DepSky even with higher latencies.



a Server times without cache

b Server times with server cache with 80% hit rate

c Server times with client cache with 80% hit rate

d Server times with client and server cache with 80% hit rate each

Figure 5.15: Server download times for test bench 3

In figures 5.14 and 5.15 we show the results for downloads with cache with 80% hit rate. We can see from figure 5.15b that using a server cache reduces the difference between RamCloud and DepSky however there is a limited influence in the client times which is expected. When using a client cache there is a similar reduction in the download times in the middleware to a server cache, but there is a much noticeable difference from the client side. In figures 5.14c and 5.15d we show the result of using both a client and a server cache, both with 80 % hit rate. Using both caches results in values very close to the use of a single cache with 100 % hit rate for the middleware server with 8.033 seconds for DepSky and 5.019 seconds for RamCloud. With 100 % hit rate in either cache these values are 0 for both backends types. They also reduce the client times by a large margin although a client cache with 100 % hit rate is still much faster which is expected as that avoids all network operations.

CHAPTER 6

CONCLUSIONS

6.1 Conclusions

The demand for cloud services, such as cloud storage solutions, is increasing constantly due to its several advantages, compared with the use of private computation and storage infrastructures and datacenters. However the migration of computations and data to the cloud leaves data vulnerable, if we consider the dependability requirements including security and privacy, as well as, reliability, availability and integrity control guarantees. Data privacy, conjugated with those guarantees under the control of users is a strong concern for many applications managing sensitive data.

Privacy-breaks and the danger of data-leakage breaking confidentiality control, can be caused by "honest but malicious" system administrators or external attackers, being today a "stopover" criteria for the generic adoption of cloud-storage services for many critical applications, for individual users but also for companies. However, complementarity, other security concerns also impose a carefully approach in the decision "to cloud or not to cloud", or "to outsource data or computations without outsource the dependability control".

While solutions exist for to address those problems they lack the ability to use the encrypted data manipulations online, making them suitable only for backup purposes or small amounts of data stored remotely in a cloud-storage solution.

In this thesis we proposed a system allowing applications to store documents, in a secure and dependable way, in a cloud-of-clouds, while allowing for online search of "always encrypted" contents and supporting information retrieval capabilities. This proposal is based on a system architecture based on a middleware approach, with the middleware services supported in two main macro-components: a client proxy, and a middleware service. The former, offers transparently the integration of searchable storage services to access "always encrypted data", distributed and replicated in a multi-cloud storage backend.

The client proxy is deployed in a client device, implementing an API for client applications, communicating transparently with the middleware server. This client proxy handles all the required data processing operations allowing the client application to simply pass documents contents through the client proxy API, supporting a multi modal interaction for search, index, store and retrieve operations. The middleware service can be deployed in different flavors: in a local server or in a computational cloud. The middleware handles the most expensive operations like training, indexing and searching, reducing the computational requirements for the client device and offering to the client a transparent access to the multi-cloud storage backend.

This is particularly relevant for resource-constrained client devices like smartphones and/or tablets or client-side devices used in Internet-of-things ecosystems supported in cloud-computing and storage solutions, in which the extra processing comes with computation, memory, storage and energy costs which can be significant or impossible to address with such devices.

One interesting aspect of our proposal is the possible adoption of different multi-cloud storage backends, by configuration. The solution can be deployed with disk based storage services in multiple and diverse clouds, as provided by typical cloud-storage services from well known providers, or "in-memory" storage services, such as multiple RAM-clouds used for reliable storage and fast access.

We implemented the proposed solution in a prototype that can be used for the intended purpose. The prototype is available for use and it is ready for the possible integration of different applications (via the provided external APIs). The implementation involves the two main macro-components: the client-component that will be used as the client-proxy (for local applications support), and the middleware service component that can be deployed in a stand-alone server or in a trusted virtualized docking cloud-based appliance, running remotely in a computational cloud.

The middleware service implements the integration of the two different variants of multi-cloud storage backend: the disk-based replicated multi-cloud data-store backend, leveraged by the Depsky solution [12], and the multi-cloud "in-memory replicated data-storage backend" supported by the implementation of the RAM-Cloud repository system model [13].

In our evaluations we concluded that the RamCloud was faster than DepSky for both upload and downloads, even when in higher latency environments, where the difference between reading from disk and reading from memory would be less noticeable when compared to the observed latency. However this improvement comes at a cost, with RamCloud being significantly more expensive to deploy than DepSky, in terms of monetary costs[1]. To have 4GB of storage available every time it would cost €144.25 per month with an on-demand contract, using for example the Amazon AWS provided services. This could be reduced to €64.56 with a three-year contract with an upfront payment, which although making it cost less per month would require an upfront payment of €2324.12. For DepSky, the same 4GB

[1]Prices were checked during October 2016

of storage cost would be €0.3 monthly, without taking into account the amount of PUT and GET requests. With 1 million PUT and GET requests for each region per month, the price would go up to €20.02 monthly. The price for both DepSky and RamCloud would increase when considering the amount of data transferred as well.

To support multi modal searching, indexing and information retrieval capabilities over the encrypted multi-cloud storage backend, among the different components of the middleware solution, we developed a Java JCA [14] compliant cryptographic-provider library (with a standard design for the generic and transparent use as any JCE-cryptographic provider). Our provider implements new cryptographic primitives for content-based searchable encryption and multi modal searchable encryption constructions [15]. And tested this support for image-similarity searches, showing that we can search images by similarity, even that the images are always maintained encrypted, thanks to the novel MIE/CBIR encryption algorithms. The results obtained from our experimental evaluation of on-line multi modal searching over multi-cloud encrypted data show that the proposal offers the desired security and privacy guarantees and provides efficient privacy-enhanced information retrieval capabilities. To achieve the observed efficiency we don't sacrifice precision and recall properties on the supported search operations and used algorithms, if compared when we use them to search and retrieve on plaintext data-stores or in encrypted data-stores.

6.2 Future Work

In this dissertation we focused primarily in providing privacy, availability and integrity of data stored in the provided multi-cloud storage backends, while allowing multi modal searches over the data-contents, without sacrificing precision and efficiency requirements. While these issues were addressed our system could be improved further in some interesting research directions. We emphasize the following ideas for some possible next steps:

First of all, we can extend the experimental evaluation initially done, to observe the system behaviors for high-scalability requirements, including the observation of multiple clients and the performance for big-data processing, possibly addressing our middleware solution as a multi-tenant service;

Replication of the middleware server could also allow to support a more exigent adversary model targeting the middleware instance as a single point of attack, for example to break the availability of the system or to cause a degradation of the service with Denial of Service attack types. This would require however the use of a replicated ecosystem of different middleware servers with coordination between the state such server instances, which could impact performance for the provided operations.

Another research line is the addition to the replication of middleware servers sharing the state allowing indexes to be split or to be migrated in runtime between different cloud-computing resources, under scale-up/scale-down conditions. On the other hand, since indexes are kept in memory there is a bound limit to the scalability of the system, even if the indexes are relatively small. Allowing the indexes to be split over several middleware

instances would solve this problem, but would require that search were coordinated between the different middleware server instances or that the client was aware of which instance it should contact for a specific search.

Finally, some implementation issues arise, in using other real setups related to the components of our solution. In particular we emphasize the performance issues of new hardware-based TPM modules, implementing the new standards in this field: TPM 1.1 and the future emergence of TPM 1.2 standardization. This integration and evaluation would be interesting to check the performance of the remote attestation and trustability auditing facilities supported in our solution that, initially, adopted a software-based emulation of the TPM functionality.

As a final interesting direction, the integration of new cryptographic primitives (added to our multi modal-searchable encryption primitives), allowing other information retrieval capabilities over multimedia documents (for example, mime-documents with different searchable media parts) could be, certainly, a hot research topic. Our work can provide an experimental base to the future implementation and integration of of such cryptographic primitives, as part of the provided search capabilities for multi modal and mobile applications accessing cloud-based multimedia documents.

BIBLIOGRAPHY

[1] Meeker, M. (2015). "Internet Trends 2015". In: *Code Conference*.

[2] Database, N. V. (2016). *"CVE Statistics"*. http://web.nvd.nist.gov/view/vuln/statistics. Ac. 04-02-2016.

[3] Chow, R., P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina (2009). "Controlling Data in the Cloud: Outsourcing Computation Without Outsourcing Control". In: *Proc. of the 2009 ACM Workshop on Cloud Computing Security*. CCSW '09. Chicago, Illinois, USA: ACM, pp. 85–90.

[4] Rushe, D. (2013). *"Google: don't expect privacy when sending to Gmail"*. http://www.theguardian.com/technology/2013/aug/14/google-gmail-users-privacy-email-lawsuit. Ac. 04-02-2016.

[5] Chen, A. (2010). *"GCreep: Google Engineer Stalked Teens, Spied on Chats"*. http://gawker.com/5637234. Ac. 04-02-2016.

[6] Greenwald, G. and E. MacAskill (2013). *"NSA Prism program taps in to user data of Apple, Google and others"*. http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data. Ac. 04-02-2016.

[7] Lewis, D. (2014). *"iCloud Data Breach: Hacking And Celebrity Photos"*. http://www.forbes.com/sites/davelewis/2014/09/02/icloud-data-breach-hacking-and-nude-celebrity-photos/. Ac. 04-02-2016.

[8] *"Facebook White Hat"* (2016). https://www.facebook.com/whitehat. Ac. 20-09-2016.

[9] F.-Brewster, T. (2015). *"Researcher Finds 'Shocking' Instagram Flaws And Ends Up In A Fight With Facebook"*. http://www.forbes.com/sites/thomasbrewster/2015/12/17/facebook-instagram-security-research-threats. Ac. 04-02-2016.

[10] Gentry, C., S. Halevi, and N. P. Smart (2012). "Homomorphic Evaluation of the AES Circuit". In: *Advances in Cryptology – CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proc.* Pp. 850–867.

[11] Ferreira, B., J. Rodrigues, J. Leitão, and H. Domingos (2014). "Privacy-Preserving Content-Based Image Retrieval in the Cloud". In: *Cornell University arXiv.org*.

[12] Bessani, A., M. Correia, B. Quaresma, F. André, and P. Sousa (2011). "DepSky: Dependable and Secure Storage in a Cloud-of-clouds". In: *Proc. of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, pp. 31–46.

[13] Ousterhout, J., A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang (2015). "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* Vol. 33. 3. New York, NY, USA: ACM, pp. 1–55.

[14] *"JCA Reference Guide"* (2016). https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html. Ac. 20-09-2016.

[15] Ferreira, B., J. Leitão, and H. Domingos (2015c). "Multimodal Searchable Encryption and the Quest for Practicality".

[16] Bozkurt, I. N., K. Kaya, A. A. Selc, and A. M. Güloglu (2008). *"Threshold Cryptography Based on Blakley Secret Sharing"*. Information Sciences.

[17] Shamir, A. (1979). "How to Share a Secret". In: *Communications of ACM*. Vol. 22. 11.

[18] Hwang, M. and T. Chang (2005). "Threshold Signatures: Current Status and Key Issues". In: *International Journal of Network Security*. Vol. 1. 3, pp. 123–137.

[19] Paillier, P. (1999). "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Eurocrypt '99*, 223–238.

[20] Gentry, C. (2009). "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proc. of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, pp. 169–178.

[21] Dijk, M., C. Gentry, S. Halevi, and V. Vaikuntanathan (2010). "Fully Homomorphic Encryption over the Integers". In: *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proc.* Pp. 24–43.

[22] Popa, R. A., E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan (2014). "Building Web Applications on Top of Encrypted Data Using Mylar". In: *Proc. of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, pp. 157–172.

[23] Goldreich, O. and R. Ostrovsky (1996). "Software Protection and Simulation on Oblivious RAMs". In: *J. ACM*. Vol. 43. 3. New York, NY, USA: ACM, pp. 431–473.

[24] Stefanov, E., M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas (2013). "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *Proc. of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, pp. 299–310.

[25] Stefanov, E. and E. Shi (2013). "Multi-cloud Oblivious Storage". In: *Proc. of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, pp. 247–258.

[26] Ferreira, B., J. Leitão, and H. Domingos (2015a). "Cifra Multimodal Indexável para Aplicações Móveis baseadas na Nuvem". In: *Proc. of Inforum'15*, pp. 386–401.

[27] Adya, A., W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer (2002). "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment". In: *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA: USENIX, pp. 1–14.

[28] Narayan, S., M. Gagné, and R. Safavi-Naini (2010). "Privacy Preserving EHR System Using Attribute-based Infrastructure". In: *Proc. of the 2010 ACM Workshop on Cloud Computing Security Workshop*. CCSW '10. Chicago, Illinois, USA, pp. 47–52.

[29] Puttaswamy, K. P. N., C. Kruegel, and B. Y. Zhao (2011). "Silverline: Toward Data Confidentiality in Storage-intensive Cloud Applications". In: *Proc. of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal, pp. 1–13.

[30] Jammalamadaka, R. C., R. Gamboni, S. Mehrotra, K. Seamons, and N. Venkatasubramanian (2008). "iDataGuard: An Interoperable Security Middleware for Untrusted Internet Data Storage". In: *Proc. of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. Companion '08. Leuven, Belgium: ACM, pp. 36–41.

[31] Rodrigues, J. (2013). "TSKY: A Dependable Middleware Solution for Data Privacy using Public Storage Clouds". MA thesis. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa.

[32] Strumbudakis, A. (2013). "FairSky: Gestão Confiável e Otimizada de Dados em Múltiplas Nuvens de Armazenamento na Internet". MA thesis. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa.

[33] Tang, Y., J. Yin, W. Lo, Y. Li, S. Deng, K. Dong, and C. Pu (2015). "MICS: Mingling Chained Storage Combining Replication and Erasure Coding". In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pp. 192–201.

[34] *"Encrypted-bigquery-client Tutorial"* (2015). https://github.com/google/encrypted-bigquery-client/blob/master/tutorial.md. Ac. 15-12-2015.

[35] Fitzpatrick, B. (2004). "Distributed Caching with Memcached". In: vol. 2004. 124. Houston, TX: Belltown Media, pp. 5–. URL: http://dl.acm.org/citation.cfm?id=1012889.1012894.

[36] Altınbüken, D. and E. G. Sirer (2012). *"Commodifying replicated state machines with openreplica"*. Tech. rep. Cornell University.

[37] Abu-Libdeh, H., L. Princehouse, and H. Weatherspoon (2010). "RACS: A Case for Cloud Storage Diversity". In: *Proc. of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, pp. 229–240.

[38] Schnjakin, M., D. Korsch, M. Schoenberg, and C. Meinel (2013). "Implementation of a secure and reliable storage above the untrusted clouds". In: *2013 8th International Conference on Computer Science Education (ICCSE)*, pp. 347–353.

[39] Schnjakin, M. and C. Meinel (2013). "Evaluation of Cloud-RAID: A Secure and Reliable Storage above the Clouds". In: *2013 22nd International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9.

[40] Chen, H. C., Y. Hu, P. P. Lee, and Y. Tang (2014b). "NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds". In: *IEEE Transactions on Computers*. Vol. 63. 1. Los Alamitos, CA, USA: IEEE Computer Society, pp. 31–44.

[41] Ling, C. W. and A. Datta (2014). "InterCloud RAIDer: A Do-It-Yourself Multi-cloud Private Data Backup System". In: *Distributed Computing and Networking*. Vol. 8314. Lecture Notes in Computer Science, pp. 453–468.

[42] Plank, J. (2013). "Erasure Codes for Storage Systems A Brief Primer". In: vol. 38. 6, pp. 7–.

[43] Jiekak, S., A.-M. Kermarrec, N. Le Scouarnec, G. Straub, and A. Van Kempen (2013). "Regenerating Codes: A System Perspective". In: *SIGOPS Oper. Syst. Rev.* Vol. 47. 2. ACM, pp. 23–32.

[44] Pereira, V. (2014). "Segurança e Privacidade de Dados em Nuvens de Armazenamento". MA thesis. Faculdade de Ciências e Tecnologias - Universidade Nova de Lisboa.

[45] Ekberg, J.-E., K. Kostiainen, and N. Asokan (2014). "The Untapped Potential of Trusted Execution Environments on Mobile Devices". In: *IEEE Security Privacy*. Vol. 12. 4, pp. 29–37.

[46] Nyman, T., B. McGillion, and N. Asokan (2015). "On Making Emerging Trusted Execution Environments Accessible to Developers". In: *Cornell University arXiv.org*.

[47] Santos, N., H. Raj, S. Saroiu, and A. Wolman (2014). "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications". In: *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, pp. 67–80.

[48] Raj, H., S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten (2015). *"fTPM: A Firmware-based TPM 2.0 Implementation"*. Tech. rep.

[49] Chen, C., H. Raj, S. Saroiu, and A. Wolman (2014a). "cTPM: A Cloud TPM for Cross-device Trusted Applications". In: *Proc. of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA, pp. 187–201.

[50] Asokan, N., J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann (2014). "Mobile Trusted Computing". In: *Proc. of the IEEE*. Vol. 102. 8, pp. 1189–1206.

[51] Stallings, W. and L. Brown (2015). "Computer Security: Principles and Pratice". In:

[52] Popa, R. A., C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan (2011). "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, pp. 85–100.

[53] Ferreira, B., J. Leitão, and H. Domingos (2015b). "Multimodal Indexable Encryption for Mobile Cloud-based Applications".

[54] *"OpenSSL"* (2016). https://www.openssl.org/. Ac. 04-08-2016.

[55] *"Jerasure"* (2016). http://jerasure.org/. Ac. 04-08-2016.

[56] *"AWS Amazon S3"* (2016). https://aws.amazon.com/s3/. Ac. 10-02-2016.

[57] *"Google Cloud Storage"* (2016). https://cloud.google.com/storage/. Ac. 10-02-2016.

[58] *"Amazon EC2"* (2016). https://aws.amazon.com/ec2/. Ac. 10-02-2016.

[59] *"RamCloud"* (2016). https://github.com/PlatformLab/RAMCloud. Ac. 23-05-2016.

[60] *"Depsky"* (2016). https://github.com/cloud-of-clouds/depsky. Ac. 23-05-2016.

[61] *"Trousers"* (2016). http://trousers.sourceforge.net/. Ac. 20-09-2016.

[62] *"TPM-Emulator"* (2016). https://github.com/PeterHuewe/tpm-emulator/. Ac. 20-09-2016.

[63] *"Microsoft Azure Blob Storage"* (2016). https://azure.microsoft.com/en-gb/services/storage/blobs/. Ac. 10-02-2016.

[64] *"Porter Stemmer"* (2016). https://tartarus.org/~martin/PorterStemmer/. Ac. 10-02-2016.

[65] *"COCOMO II - Constructive Cost Model"* (2016). http://csse.usc.edu/tools/COCOMOII.php. Ac. 20-09-2016.

[66] *"COCOMOII: Model Definition Manual"* (2016). http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf. Ac. 20-09-2016.